TESTING THE REFACTORING ENGINE OF THE NETBEANS IDE

BY

KELY M. GARCIA

B.S., Pontificia Universidad Catolica del Peru, 1998

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois

Abstract

Refactorings are behavior preserving program transformations that improve program design. Refactoring engines are tools that automate the application of refactorings: first the user chooses a refactoring to apply, then the engine checks if the transformation is safe, and if so, transforms the program. Refactoring engines are a key component of modern IDEs and programmers rely on them to perform refactorings. A bug in the refactoring engine can have severe consequences as it can erroneously change large bodies of the program.

The purpose of this thesis is testing several refactorings in NetBeans, a popular IDE that includes a refactoring engine for the Java programming language. We use two types of tests: (1) existing tests manually written for Eclipse, another popular IDE for Java, and (2) tests automatically generated with a framework called ASTGen. We reported 53 new bugs for the NetBeans refactoring engine; 27 bugs were found using Eclipse tests, and 26 bugs were found using ASTGen tests. We reported these bugs to Issuezilla, the bug-tracking system for NetBeans, and the developers already fixed 5 of these bugs, declared 7 bugs as duplicates, declared 1 bug as "won't be fixed", declared 2 bugs as "work fine in latest version", and confirmed 43 as real bugs.

To my loving husband Roberto, my extended family, and my close friends.

Thanks to all of you for pushing and helping me to reach my goals.

Acknowledgments

It is a pleasure to thank the many people who made this thesis possible. First, I want to thank the Fulbright Commission in Peru and the Department of Computer Science at the University of Illinois at Urbana Champaign for their financial support during my master studies.

My sincere thanks to my advisor, Prof. Darko Marinov, for suggesting me the idea of testing refactoring engines and for his valuable feedback regarding the researching and writing of this master thesis.

I want to thank my colleagues in Siebel Center: Brett Daniel, Choonghwan Lee, Danny Dig, and Steve Lauterburg who contributed with interesting ideas and relevant critiques for this thesis. Special thanks to Danny Dig and Brett Danniel with whom I worked with in the project called ATRE. I also want to thank my friends at UIUC: Chi Trinh, Dmitry Yershov, Hengzhi Zhong, Kashif Manzoor, Luis Rios, Marco Milla, Nidhi Doshi, and Pablo Reyes.

I want to thank to my close friends from high school Betty and Rocio and my close friends during my undergrad Carlos, Rocio, and Victor. Your friendship is one the most valuable things for me.

My parents have always been an inspiration in my life. They have always supported my dreams and aspirations. I want to thank you both for all your love, patience, and confidence. Thanks also to my brothers Cristian and Jesus for their encouragement and support.

Most importantly, I wish to thank my husband, Roberto Lavarello, for his love, support, and encouragement. You are my sunshine.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Refactoring is a disciplined technique of applying behavior preserving transformations to improve program design [1, 2]. Refactoring is gaining popularity, as evidenced by the inclusion of refactoring engines in modern IDEs for Java such as NetBeans (`http://www.netbeans.org`) or Eclipse (`http://www.eclipse.org`). Refactoring is also a key practice of agile software development methodologies, such as eXtreme Programming [3], whose success prompts even more developers to use refactoring engines on a regular basis. Indeed, the common wisdom views the use of refactoring engines as one of the safest ways of transforming a program, since manual refactoring is error-prone [1, 4].

This chapter provides an overview of this thesis. It first lists some basic terminology (Section 1.1). It then briefly explains the problem of testing refactoring engines (Section 1.2) and our proposed solution (Section 1.3). It next shows an example of how we tested the PullUpMethod refactoring in NetBeans using Eclipse and ASTGen tests (Section 1.4). It finally summarizes the results of our work on testing the refactoring engines in NetBeans (Section 1.5).

1.1 Terminology

This section defines some relevant concepts that will be used throughout this document.

- **Refactoring:**

  This is a disciplined technique of applying behavior-preserving transformations to a program with the intent of improving its design [1]. Each program experiences many changes during its lifetime due to different factors such as adding new requirements, changing existing requirements, fixing bugs, or a new development team having a hard time trying to understand the inherited

program. Some of these changes lead to modifications in the functionality and behavior of the program, but others are refactorings that make no modifications in behavior.

Each refactoring has a name, a number of parameters, a set of preconditions, and a set of specific transformations to perform [5]. For instance, there is a refactoring called PullUpMethod which moves a method from one class to one of its superclasses. The aim of this refactoring is to centralize functionality in a common superclass. The number of refactoring parameters depends on the refactoring. For example, the PullUpMethod refactoring parameters are: the method to pull up, the superclass, and the optional parameter `Make Abstract`. Figure 1.1 shows an example of the PullUpMethod refactoring.

Refactoring: PullUpMethod C.m() to superclass A

Before:

```
A.java
public class A {
}
class B extends A {
}
class C extends B {
  void m(){}
}
```

After:

```
A.java
public class A {
  protected void m(){}
}
class B extends A {
}
class C extends B {
}
```

Figure 1.1: Example for PullUpMethod

These are several PullUpMethod preconditions:

- The superclass must exist.
- The method must not exist in the superclass.
- The method implementation must not call methods inaccessible in the superclass.
- The method implementation must not access variables inaccessible in the superclass.
- Overridden methods in direct subclasses of the superclass that have a method signature equal to the selected method must have a return type equal to the selected method return type.

The PullUpMethod refactoring without setting `Make Abstract` performs the following transformations:

- Removes the selected method in the origin class.

- Creates the selected method in the superclass.

- **Refactoring engines:**

  These are tools that automate the application of refactorings. The programmer selects which refactoring to apply, and the refactoring engine will automatically apply the transformations across the entire program if the preconditions are satisfied. Figure 1.2 shows that the refactoring engine takes an input program and some refactoring parameters, and first checks whether the program satisfies the preconditions in order to enable the execution of the transformations. If any precondition is not satisfied, then the refactoring engine will return a list of error messages; otherwise, the refactoring engine will appropriately change the program and produce the refactored program.

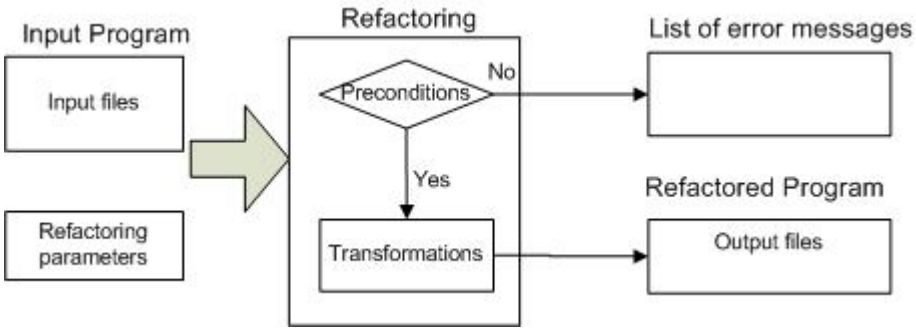

Figure 1.2: Refactoring process

- **Refactoring preconditions:**

  These are assertions that the input program must satisfy for the refactoring to be behavior-preserving [2]. They describe the restrictions that need to be satisfied before applying the refactoring.

- **Refactoring error message:**

  These are messages related to violations of refactoring preconditions. When a test does not satisfy the preconditions, the refactoring will return a list of error messages. These messages can be warnings or errors. In this thesis, we will refer to both kinds of messages as *error messages* because we treat them in the same way.

  For example, these are two PullUpMethod error messages from NetBeans:

  - Cannot pull up any members. The selected type has no supertypes in the currently opened projects (Violates the precondition that the superclass must exist.)
  - Selected method already exists in the target type (Violates the precondition that the method must not exist in the superclass.)

  For example, this is a PullUpMethod warning message from NetBeans:

  - The visibility of the selected method will be changed to default (Does not violate any precondition but will lead to some program change that the programmer may not expect.)

- **Test:**

  This term is used in this thesis to refer to all the required inputs necessary to perform a refactoring. In that sense, a test includes the input program, the refactoring name, and the refactoring parameters. As a convention, we use a more specific term such as *input program* or *refactoring parameter* when it is unclear from the context what the term *test* refers to.

- **ASTGen:**

  This is a library that allows developers to write imperative generators whose executions produce input programs for refactoring engines [6]. More precisely, ASTGen provides a library of generic, reusable, and composable generators that produce Abstract Syntax Trees (ASTs). Using ASTGen, a developer can focus on the creative aspects of testing rather than the mechanical production of test inputs. Instead of manually writing input programs, a developer writes a generator whose execution produces thousands of programs with structural properties that are relevant for the specific refactoring being tested. For example, to test the PullUpMethod refactoring, the input program should have classes that have inheritance relationships where the

subclass declares a method and this method is referenced in several classes in the hierarchy. ASTGen generators can systematically produce a large number of programs that satisfy such constraints.

- **Differential testing:**

  This technique addresses a known testing problem, called the *oracle problem*, which refers to evaluating correctness of test results [7]. Differential testing requires two or more similar systems like Java IDE and some test inputs to perform on these systems. Differential testing requires that the same inputs be performed on both systems and then their results compared. To test refactoring engines, we run the same input programs on both NetBeans and Eclipse and then compare the output from the two engines, either refactored programs or error messages.

- **AST Comparator:**

  This program compares NetBeans and Eclipse output files [6]. The comparison first normalizes two abstract syntax trees by sorting the methods and fields by name, and then compares the method bodies and field expressions of the appropriate pairs of methods and fields.

- **Refactoring runner:**

  This is a class in NetBeans that performs the refactoring for each input program (using the appropriate parameters) from the existing Eclipse tests or automatically generated ASTGen tests. When we perform a refactoring, the result can be either a list of error messages or a refactored program. This class contains the code that creates new refactoring objects (with appropriate parameter values) and invokes the main "refactor" method on them.

- **Program element:**

  This is a general term used to refer to elements of a program. Some examples of program elements are: packages, classes, interfaces, methods, fields, and local variables.

- **Type:**

  This is a general term used to refer to a Java class or a Java interface. The error messages in NetBeans and Eclipse often include this term.

1.2 Thesis problem

It is important that refactoring engines be reliable: a bug in a refactoring engine can silently introduce bugs in the refactored program and lead to difficult debugging sessions. If the input program compiles but the refactored program does not, the refactoring is obviously incorrect and can be easily undone. However, if the refactoring engine erroneously produces a refactored program that compiles but does not preserve the semantics of the input program, this can have severe consequences.

Since refactoring engines are very complex and must be reliable, developers of refactoring engines have invested heavily in testing. For example, NetBeans version 6.0 M3 has over 250 XTest tests for refactorings, and Eclipse version 3.2 has over 2,600 unit tests for refactorings (publicly available from the Eclipse CVS repository). Conventionally, testing a refactoring engine involves creating input programs by hand along with their expected outputs, each of which is either a refactored program or an expected error message. The developers then execute these tests automatically with a tool such as XTest or JUnit [8]. Writing such tests manually is tedious and results in incomplete test suites, potentially leaving many hidden bugs in refactoring engines.

1.3 Proposed solution

Our goal is to find and report new bugs to help the developers of refactoring engines to improve their reliability. This thesis involves testing six refactorings in NetBeans. Two types of tests were used for this purpose: (1) existing tests manually written for Eclipse and (2) tests automatically generated with ASTGen. Reusing Eclipse tests in NetBeans allowed us to find similar bugs in NetBeans to those previously reported in Eclipse. Using ASTGen tests allowed us to complement Eclipse tests because the generated tests cover different scenarios for the selected refactorings.

We first made some changes to NetBeans to run Eclipse and ASTGen tests. We wrote some test generators using ASTGen to automatically generate tests. We next wrote refactoring runners for the selected refactorings. We then ran Eclipse and ASTGen tests in NetBeans using our runners,

and we also run ASTGen tests in Eclipse. We found bugs using differential testing and several other oracles. We finally reported these bugs to Issuezilla, the bug-tracking system used for the NetBeans project. Our results include reporting 53 new bugs, and they are publicly available online from the ASTGen web page at `http://mir.cs.uiuc.edu/astgen`.

1.4 Example

This section first explains how we tested the PullUpMethod refactoring using Eclipse and ASTGen tests. It then includes the list of bugs found for PullUpMethod using Eclipse and ASTGen tests. It shows one relevant bug found using Eclipse tests and one relevant bug found using ASTGen tests.

1.4.1 PullUpMethod

In order to test the PullUpMethod refactoring, we first executed our runners using as tests the existing 95 input programs from Eclipse tests. Next, we wrote an ASTGen test generator which generated 1312 input programs for PullUpMethod and executed our runners using as tests these generated input programs. We finally found 8 bugs using differential testing and several oracles; 5 bugs were found using Eclipse tests, and 3 bugs were found using ASTGen tests. We reported these bugs to Issuezilla (`http://www.netbeans.org/community/issues.html`), the bug-tracking system used for the NetBeans project.

1.4.2 Bugs found using Eclipse tests

We found 5 bugs for PullUpMethod using Eclipse tests. They correspond to bugs 111563, 111564, 111565, 111566, and 111567 in Issuezilla. Table 1.1 shows bug 111563, where the bug number refers to the number used in Issuezilla.

Bug: #111563
Refactoring: PullUpMethod B.m() to superclass A
Problem: Compilation error: cannot find symbol: method a
Solution: It should return a message such as: "B.a referenced in one of the moved elements is not accessible from A"

Before:

```
A.java
class A {}
class B extends A {
  void m() {
    a();
  }
  private void a(){}
}
```

After:

```
A.java
class A {
  void m() {
    a();
  }
}class B extends A {
  private void a(){}
}
```

Table 1.1: Bug #111563 for PullUpMethod

1.4.3 Bugs found using ASTGen tests

We found 3 bugs for PullUpMethod using ASTGen tests. They correspond to bugs 108486, 108487, and 108488 in Issuezilla. Table 1.2 shows the bug 108486.

Bug: #108486
Refactoring: PullUpMethod B.m() to superclass A
Problem: Compilation errors: m(Object) cannot be applied to m() and cannot find symbol: variable theField
Solution: 'super.theField' should have been changed to 'this.theField'

Before:

```
A.java
public class A {
    private Object theField;
    class B extends A {
        private void m(){
            super.theField=null;
        }
        void m(Object newArg){
            B.this.m();
        }
    }
}
```

After:

```
A.java
public class A {
    private Object theField;
    class B extends A {
        void m(Object newArg){
            B.this.m();
        }
    }
    private void m() {
        super.theField = null;
    }
}
```

Table 1.2: Bug #108486 for PullUpMethod

1.5 Summary

We tested 6 refactorings in NetBeans using Eclipse and ASTGen tests. We used NetBeans version 6.0 M3 to run our tests and reported 53 bugs. 27 bugs were found using Eclipse tests, and 26 bugs were found using ASTGen tests. We manually replicated those bugs that could be performed in NetBeans version 6.0 M9 before reporting them to Issuezilla. The developers of NetBeans already fixed 5 of these bugs, declared 7 bugs as duplicates, declared 1 bug as "won't be fixed", declared 2 bugs as "work fine in latest version", and confirmed 43 as real bugs.

Chapter 2

Refactorings in NetBeans

This chapter explains how we run the refactoring tests in NetBeans and the procedure used to find bugs in NetBeans. It first provides a list of refactorings (Section 2.1). It then shows relevant information about refactorings (Section 2.2) and explains how to run refactoring tests in NetBeans (Section 2.3). It next lists the refactoring runners in NetBeans (Section 2.4). It finally describes the list of oracles (Section 2.5) and the procedure used to find bugs (Section 2.6).

2.1 List of refactorings

This section first lists refactorings in NetBeans version 6.0 M3. It then explains the four refactorings groups into which all these refactorings can be divided.

2.1.1 List of refactorings in NetBeans

NetBeans version 6.0 M3 implements the following refactorings:

- **Rename:**

  This refactoring renames a program element.

- **ExtractMethod:**

  This refactoring creates a new method using the selected statements and replaces these statements with a method call.

- **ChangeMethodParameters:**

  This refactoring allows to add parameters, remove parameters, and change the type of existing parameters.

- **EncapsulateFields:**

  This refactoring allows to replace one or more fields using getter and setter methods.

- **PullUp:**

  This refactoring allows to pull up a method or a field from a subclass to its superclass.

- **PushDown:**

  This refactoring allows to push down a method or a field from a class to its subclasses.

- **MoveClass:**

  This refactoring allows to move a class to another package.

- **MoveClassInnerToOuterLevel:**

  This refactoring allows to move an inner class to an outer or separate class.

- **ConvertAnonymousClassToInner:**

  This refactoring allows to convert an anonymous or local class to an inner class.

- **ExtractInterface:**

  This refactoring allows to create a new interface with a set of methods and make the selected class to implement it.

- **ExtractSuperclass:**

  This refactoring allows to create a common superclass for the selected classes.

- **UseSupertypeWherePossible:**

  This refactoring allows to replace occurrences of a type with one of its supertypes.

2.1.2 Groups of refactorings

According to Enns [8], refactorings implemented in NetBeans and Eclipse can be divided in four groups. The first three groups are based on the program changes that the refactoring makes; these changes can be to the physical structure of the program, to the class level, or inside a class. We refer to these refactorings as *core refactorings*. The fourth group of refactorings either use history information about performed refactorings or involve performing more than one refactoring (e.g.,

Undo or CreateScript). For the purpose of this thesis, we do not consider the fourth group because we are interested in testing each refactoring in isolation. While NetBeans version 6.0 M3 includes 12 core refactorings, Eclipse version 3.2 includes 21 core refactorings. The following tables list equivalent refactorings in NetBeans and Eclipse for each groups of refactorings.

Group 1: Physical Structure

These refactorings impact the physical structure of code and classes. Table 2.1 shows refactorings of this group in NetBeans and Eclipse. Refactorings that are in the same row provides similar functionality but not necessary the same. For instance, while Move refactoring in Eclipse can be applied over several program elements— including instance methods, static fields, and types— MoveClass refactoring in NetBeans can only be applied to types.

NetBeans	**Eclipse**
Rename	Rename
MoveClass	Move
ChangeMethodParameters	ChangeMethodSignature
ConvertAnonymousClassToInner	ConvertAnonymousClassToNested
MoveClassInnerToOuterLevel	ConvertMemberTypeToTopLevel

Table 2.1: Refactorings of group 1

Group 2: Class Level Structure

These refactorings have an impact with respect to the class level structure. Table 2.2 shows refactorings of this group in NetBeans and Eclipse. NetBeans has no refactoring similar to the GeneralizeDeclaredType and InferGenericTypeArguments in Eclipse. When an Eclipse refactoring does not have a similar in NetBeans, we put `n/a` in the NetBeans column.

NetBeans	**Eclipse**
PushDown	PushDown
PullUp	PullUp
ExtractInterface	ExtractInterface
ExtractSuperclass	ExtractSuperclass
UseSupertypeWherePossible	UseSupertypeWherePossible
n/a	GeneralizeDeclaredType
n/a	InferGenericTypeArguments

Table 2.2: Refactorings of group 2

Group 3: Structure inside a class

These refactorings only have an impact inside its class. Table 2.3 shows refactorings of group 3 in NetBeans and Eclipse. For this group, Eclipse has 7 more refactorings.

NetBeans	**Eclipse**
EncapsulateFields	EncapsulateField
ExtractMethod	ExtractMethod
n/a	ExtractConstant
n/a	ExtractLocalVariable
n/a	ConvertLocalVariableToField
n/a	IntroduceParameter
n/a	IntroduceFactory
n/a	IntroduceIndirection
n/a	Inline

Table 2.3: Refactorings of group 3

Group 4: Refactoring support

These refactorings use history refactoring information and could perform many refactorings at the same time. Table 2.4 lists refactorings of group 4 in NetBeans and Eclipse. We do not consider these refactorings further in this thesis.

NetBeans	**Eclipse**
Undo	Undo
Redo	Redo
n/a	MigrateJarFile
n/a	CreateScript
n/a	ApplyScript
n/a	History

Table 2.4: Refactorings of group 4

2.2 Relevant information about refactorings

This section provides information about refactoring tests. It first includes a list of differences between the existing tests for NetBeans and Eclipse refactoring engines, then a list of differences between Eclipse tests and ASTGen tests, and finally relevant information about refactorings in NetBeans and Eclipse. This information applies for the NetBeans version 6.0 M3 that we worked with, but there are big changes for NetBeans 6.0 M10, and in the future NetBeans may include runners for ASTGen [9].

2.2.1 Differences between tests for NetBeans and Eclipse refactoring engines

There are some differences between tests for NetBeans and Eclipse refactoring engines such as:

- **Refactoring engine tests:**

  Eclipse performs refactoring tests in this way: for each test, it creates a project at runtime, copies the input files for this test to the project, performs the refactoring, and then closes this project. Closing the projects after performing the refactoring tests avoids running out of memory when dealing with a large number of projects.

  NetBeans performs refactoring tests in this way: it loads all input files for the tests that will be run into one project, then it performs each test, and finally it creates a summary file that shows the status of each test. Using one project to perform all refactoring tests can make NetBeans to run out of memory when there are many input files for the tests. In addition, it requires that different tests have input files with different (fully qualified) names.

- **Execution time:**

  Running refactoring tests in NetBeans with our runners takes approximately 2 times more than running the same tests in Eclipse. We got this ratio running Eclipse tests in NetBeans version 6.0 M3 and Eclipse version 3.2. However, this ratio increases when we use the generated tests because a new environment is launched more than once in NetBeans.

2.2.2 Differences between Eclipse tests and ASTGen tests

There are some differences between Eclipse and ASTGen tests such as:

- **Refactoring tests parameters:**

  For all input programs generated by one ASTGen generator, we want to run the same refactoring with the same refactoring parameters. However, for Eclipse tests, not all input programs are run with the same parameters. Rather, Eclipse test suites have methods that run refactorings with different parameters on different input programs. For instance, while the refactoring tests parameters for an ASTGen generator for RenameClass could be rename class A to B, the refactoring tests parameters for an Eclipse test suite for RenameClass could be rename class A to B for some tests and rename class A to C for other tests. These differences mean that our NetBeans runners

for Eclipse tests need to be more complex than our runners for ASTGen tests.

- **Package:**

  While ASTGen tests have as input files classes either declared in package `p` or classes declared in the default package, Eclipse tests have as input files classes declared in various packages. A limitation for our current implementation to run refactoring tests in NetBeans is that we have to change the package statement for all input files from Eclipse and ASTGen tests that are copied to the `work` directory, which must be done due to the way that NetBeans performs the refactoring tests, all at once in the same project. Because we do not update import statements or any other statements that reference the package, there were some cases where the AST Comparator found differences between the output files that are false positives.

2.2.3 Relevant information about refactorings in NetBeans and Eclipse

This section provides detailed information about refactorings that was used to prioritize our work. Table 2.5 shows relevant information for refactorings. The first column lists the refactorings and their program elements. The second column shows the first NetBeans version that provided this refactoring. The next two columns show the number of bugs and the number of tests for this refactoring in Eclipse. The last column shows the first Eclipse version that provided this refactoring. NetBeans and Eclipse version indicates the first time that the refactoring was included; however, some refactorings have been extended after that time.

The bugs were obtained from Bugzilla (`https://bugs.eclipse.org/bugs/`), the bug-tracking system system used for Eclipse. Bugs related to new features or user interface were not considered for this study. The search criteria was the following:

- bugs modified during 10-22-2005 and 10-24-2006

- product is `Eclipse JDT`

- status equal to `New`, `Assigned`, `Reopened`, `Resolved`, `Verified`, `Closed`

- resolution different from `INVALID`, `DUPLICATE`, `LATER`

- the bug summary should contain any of the words for the selected refactoring.

2.3 Running refactorings in NetBeans

This section includes the list of changes that we made to NetBeans to run Eclipse and ASTGen tests. It first explains the refactoring tests organization. It then lists the changes made to NetBeans. It next lists the configuration files. It finally explains how to run refactoring tests in NetBeans.

2.3.1 Refactoring tests organization

Eclipse uses several structures to store its refactoring tests. In particular, two of them are used to store the tests of the selected NetBeans refactorings. For convenience, ASTGen tests also use one of these two structures. This section explains the two structures mentioned above.

NetBeans version 6.0 M3 includes some refactoring tests that are organized as follows. The `src` directory includes all the refactoring test suites, the `projects` directory includes the input files, and the `goldenfiles` includes the expected output files. NetBeans uses a work directory called `data`. Every time we run the refactoring tests in NetBeans, the input files are copied to the `data` directory. We made some changes to NetBeans in order to copy the input programs from Eclipse and ASTGen tests to the working directory and to copy the refactored programs to the output directory for Eclipse and ASTGen tests. Our runners include the logic to work with the following two Eclipse structures:

Structure #1

Figure 2.1 shows the first structure. The Eclipse test suite is 'RenameNonPrivateField', and it includes several test directories. Each test directory has three subdirectories: the `in` directory has the input files, the `out` directory has the output files from Eclipse, and the `outNB` has the output files from NetBeans.

Structure #2

Figure 2.2 shows the second structure. The Eclipse test suite is 'SefTests'. In this case several tests

	NetBeans	Eclipse		
Refactoring	**Version**	**# Bugs**	**# Tests**	**Version**
Rename (Method) Rename (Parameter) Rename (Field) Rename (Local variable) Rename (Type) Rename (Type parameter) Rename (Enum Constant) Rename (Compilation unit) Rename (Package) Rename (Source folder) Rename (Project)	4.1	96	754	1.0
Move (Instance method) Move (Static method) Move (Static field) Move (Type) Move (Compilation unit) Move (Package) Move (Source folder) Move (Project)	4.1	78	223	2.1
ExtractSuperclass	5.0	22	5	3.2
ExtractInterface	5.0	22	126	2.1
UseSupertypeWherePossible	5.0	6	110	3.1
PushDown (Method) PushDown (Field)	5.0	9	95	2.1
PullUp (Method) PullUp (Field) PullUp (Member type)	5.0	29	139	2.1
ChangeMethodSignature	4.1	33	127	2.1
ExtractMethod	5.0	37	389	1.0
ExtractLocalVariable	No	22	127	2.1
ExtractConstant	No	5	58	2.1
Inline (Method) Inline (Local variable) Inline (Constant)	No	42	240	2.1
ConvertAnonymousClassToNested	5.0	11	43	2.1
ConvertMemberTypeToTopLevel	5.0	0	0	2.1
ConvertLocalVariableToField	No	3	52	2.1
IntroduceIndirection	No	10	31	3.2
IntroduceFactory	No	3	44	3.1
IntroduceParameter	No	16	20	3.1
EncapsulateField	4.1	5	32	2.1
GeneralizeDeclaredType (Type reference) GeneralizeDeclaredType (Field) GeneralizeDeclaredType (Local variable) GeneralizeDeclaredType (Parameter)	No	7	0	3.1
InferGenericTypeArguments (Projects) InferGenericTypeArguments (Packages) InferGenericTypeArguments (Types)	No	27	58	3.1
Total			**2,673**	

Table 2.5: Relevant information about refactorings

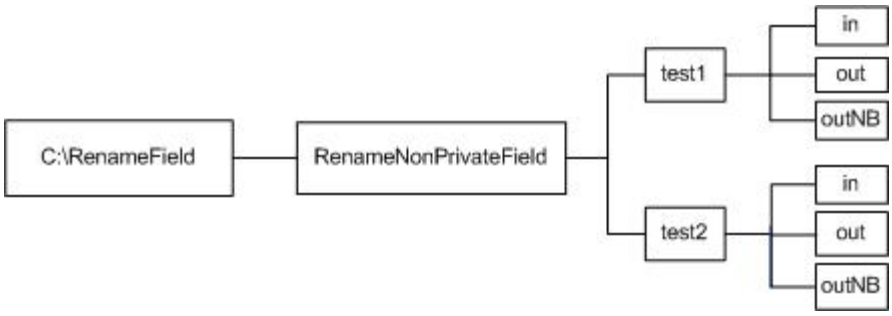

Figure 2.1: Structure #1

share the same directory to store their input or output files. For instance, 'base_in' includes the input files for 7 tests and 'object_in' includes the input files for 18 tests. This structure does not allow tests with more than one input file. For that reason, ASTGen tests use the first structure.

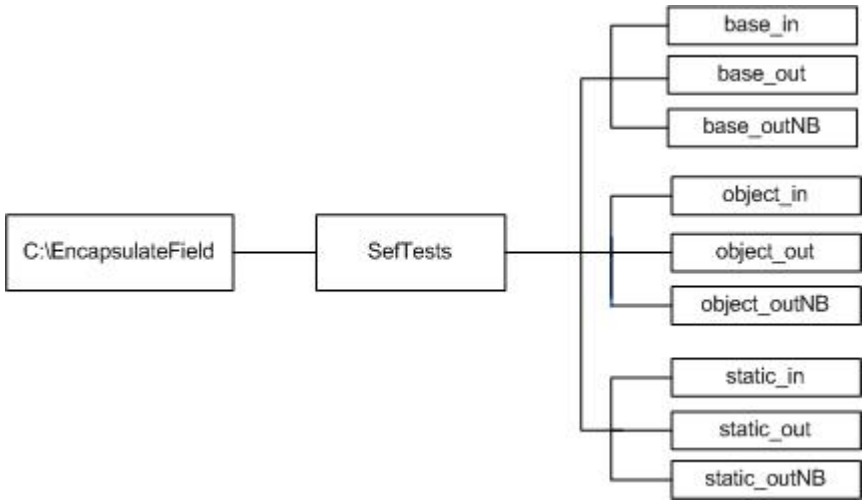

Figure 2.2: Structure #2

2.3.2 Changes made to NetBeans

In order to perform the refactorings in NetBeans using Eclipse tests and ASTGen tests, it was necessary to make changes to some refactoring classes in NetBeans.

We added code required to store information about the status and error messages for each test from a refactoring test suite. Figure 2.3 shows the superclass RefactoringTestCase, the class RenameTestCase, and three refactoring test suites for Rename. An example of a runner for AST-

Gen tests is `RenameField_Dualclassfieldreference`, an example of a runner for Eclipse tests is `RenameTypeParameter`, and an example of an existing NetBeans test is `SimpleTest`. NetBeans tests were used as a basis to create our runners. The following is the list of the modified and added files.

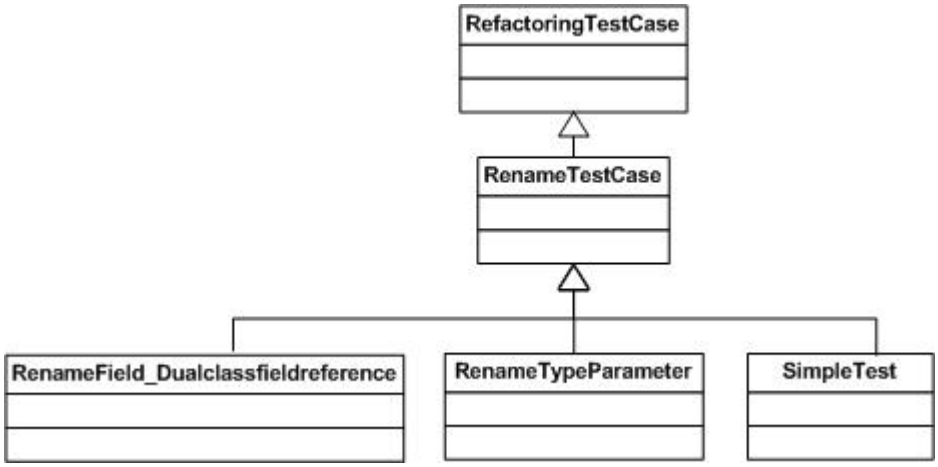

Figure 2.3: Refactoring class diagram

- **Modified class RefactoringTestCase**

  This is the superclass for all refactoring test suites in NetBeans. The refactoring runners for Eclipse tests and ASTGen tests also have this class as its superclass. We added methods that read information from the configuration files and copy a group of input files to the work directory. Due to the way that NetBeans performs refactorings, it was necessary to have a method that could copy a specific number of input files to avoid running out of memory. We modified the `tearDown` method to copy all output files to the corresponding Eclipse tests or ASTGen tests directory and to write a summary of the results to a `log` file.

- **Modified class LogTestCase**

  This class provides some methods to write to the `log` file. It also includes the `tearDown` method that compares the NetBeans output files against the expected output files. We commented the lines that perform this comparison because we do not have the expected output files.

- **Modified RefactoringTestCase's subclasses**

  We modified immediate RefactoringTestCase's subclasses such as RenameTestCase and PushDownTestCase in order to update the status and refactoring error messages returned by NetBeans

for each test.

- **Added NetBeans refactoring runners**

  We added refactoring runners in NetBeans to run Eclipse tests and ASTGen tests. There is a NetBeans refactoring runner for each selected Eclipse test suite, and ASTGen test generator.

2.3.3 Configuration files

There are mainly three configuration files we used to run the refactoring tests in NetBeans. We added two of these files to run our refactoring runners. The first file is a common configuration file for all runners, and the second file is a specific configuration file for each runner.

- The common configuration file called `Common.args` includes global variables that specify:
  - NetBeans work directory for refactoring tests

    e.g., C:\1_Workspace\NetBeans 6.0 M3\refactoring\test\work\sys\data \projects\default\src
  - Input directory for ASTGen tests

    e.g., C:\Temp
  - Input directory for Eclipse tests

    e.g., C:\Temp\Eclipse_Tests
  - The output files directory name for NetBeans

    e.g., outNB
- The specific configuration file `.args` required by each runner specify the following parameters:
  - Project name that specifies the selected project name inside the work directory

    e.g., default
  - Refactoring name directory

    e.g., renamefield
  - Test directory name

    e.g., singleclassfieldreference0

- The input file for the refactoring

  e.g., A.java

- Number of tests to be run

  e.g., 90

- Group of tests to be run

  e.g., 1

- NetBeans uses an `.xml` configuration file which specifies the runners to be executed by NetBeans. For instance, the following lines specify to execute the runner `RenameMethodInInterface`

```
<testbag testattribs="full,jdk15" executor="ide" name="Rename">
 <testset dir="qa-functional/src">
  <patternset>
   <include name="org/netbeans/test/refactoring/rename/RenameMethodInInterface.class"/>
  </patternset>
 </testset>
</testbag>
```

2.3.4 How to run refactoring tests in NetBeans

There are two ways to execute our runners:

- **Via command line:** go to the directory `..\netbeans60m3\refactoring\test` and type: `ant runtests -Dxtest.testtype=qa-functional`

- **Via graphical user interface:** In NetBeans select the project "Refactoring" then right click on it, and select `XTest\Run qa-functional tests`

After executing the runners, there will be a `log` file for each runner that provides summary and detailed information about the tests that were run.

2.4 List of refactoring runners

This section includes the list of selected refactorings to test using Eclipse and ASTGen tests and the list of refactoring runners in NetBeans.

Based on the information included in Table 2.5 such as the number of bugs reported in Eclipse for each refactoring and when the refactoring was first introduced in NetBeans, we selected to test the following NetBeans refactorings:

- Rename
- MoveClassInnerToOuterLevel
- PullUp
- PushDown
- EncapsulateField
- ChangeMethodParameters

The last refactoring is tested only with ASTGen tests. Table 2.6 shows the refactoring runners for each of the selected refactorings. The first column indicates the type of test, the second column lists the refactoring, and the last column lists the refactoring runner.

2.5 Oracles

An important problem in automated generation of test inputs is automated checking of outputs, also known as the *oracle problem*. A seemingly ideal oracle for a refactoring engine would tell whether an input program and its refactored version have the same semantics. However, checking that two programs have the same semantics is undecidable in general. Moreover, even if the two programs have the same semantics, the refactoring engine might not have performed the required changes on the program. This section describes a variety of oracles that we used to validate the semantics of the refactored program.

The simplest oracles check that the refactoring engine does not crash (i.e., does not throw an uncaught exception) and that the refactored program compiles. We use differential testing (Section 1.1) in which one implementation serves as the oracle for another implementation. Specifically, we run the same input programs on NetBeans and Eclipse and compare their refactored programs

Type	NetBeans refactoring	Runner in NetBeans
ASTGen	ChangeMethodParameters	Methodwithparameterreference
		Singleclassmethodreference
ASTGen	EncapsulateFields	Classwitharrayfield
		Dualclassfieldreference
		Singleclassfieldreference
ASTGen	MoveClassInnerToOuterLevel	Classrelationships
		Dualclassfieldreference
ASTGen	PullUp	DualClassFieldReference
		DualClassMethodChild
		TripleClassChildDecl
		TripleClassMethodDecl
ASTGen	PushDown	DualClassFieldReference
		DualClassMethodParent
ASTGen	Rename	RenameMethod_Singleclassmethodreference
		RenameField_Singleclassfieldreference
		RenameField_Dualclassfieldreference
		RenameClass_AllRelationships
Eclipse	EncapsulateFields	Encapsulate
Eclipse	MoveClassInnerToOuterLevel	MoveInnerToTopLevel
Eclipse	PullUp	PullUp
Eclipse	PushDown	PushDown
Eclipse	Rename	RenameMethodInInterface
		RenameMethods
		RenameVirtualMethodInClass
		RenameStaticMethod
		RenameNonPrivateField
		RenamePrivateField
		RenamePrivateMethod
		RenameType

Table 2.6: Refactoring runners

or error messages.

The oracles help us to identify differences in the possible groups for the refactored programs in NetBeans and Eclipse. Figure 2.4 shows the four groups for the refactored programs. When a program is "not refactored", it means that the refactored engine returned an error message. We analyzed differences in groups: NB, EC, and BOTH. Refactored programs in these groups could have output files that compile or not compile in NetBeans and Eclipse.

- **Tests not refactored in either NetBeans or Eclipse (NONE):** These are tests for which both NetBeans and Eclipse returned an error message.

- **Tests only refactored in NetBeans (NB):** These are tests for which NetBeans generated a refactored program, but for which Eclipse returned an error message. These tests can either

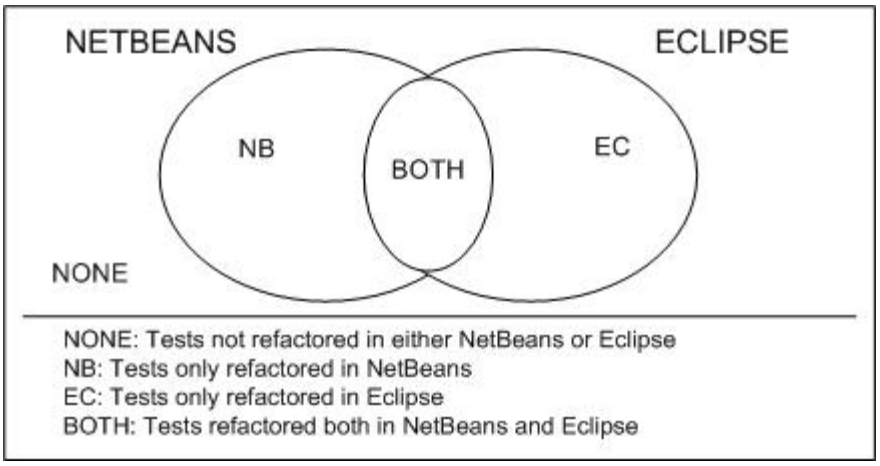

Figure 2.4: Summary Results

imply a bug or that NetBeans and Eclipse have different preconditions for the refactoring.

- **Tests only refactored in Eclipse (EC):** These are tests for which Eclipse generated a refactored program, but for which NetBeans returned an error message. These tests can either imply a bug or that NetBeans and Eclipse have different preconditions for the refactoring.

- **Tests refactored both in NetBeans and Eclipse (BOTH):** These are tests for which NetBeans and Eclipse generated a refactored program. We applied differential testing on the output files to find bugs.

All oracles apply to all tests except for those where a difference is explained. These are the implemented oracles:

- **Do crash oracle:**

  Our simplest oracle checks that the refactoring engine does not throw an uncaught exception. Such an oracle is often used as a sanity check in "smoke testing".

- **Do not compile oracle:**

  Our next oracle checks that the refactored program compiles. We filter all test programs through the compiler and pass to the refactoring engine only those input programs that compile. A correct refactoring should thus always produce refactored programs that compile. If any of the output files for a particular test do not compile after being refactored, then it means that there was a bug in the refactoring engine.

Among Eclipse tests that yield compilation errors, there can be tests that were expected to return an error message and tests that were expected to generate a refactored program. For this reason, this oracle includes two lists of compilation errors:

- List of different compilation errors for tests expected to return an error message
- List of different compilation errors for tests expected to generate a refactored program

- **Erroneously refactored oracle:**

  This oracle checks that tests expected to return an error message indeed return an error message. There are some Eclipse tests that are expected to return an error message. However, some of these tests can erroneously generated refactored programs in NetBeans without exposing any compilation error. We have manually performed these tests in Eclipse to determine the error message that each of them returns in Eclipse; the tests themselves only assert that the Eclipse refactoring engine should return some error message and not proceed with the refactoring.

- **Not refactored oracle:**

  This oracle checks that tests expected to produce a refactored program instead return an error message and also that tests expected to return an error message instead produce a refactored program. Usually tests for which one refactoring engine generates a refactored program, and for which another refactoring engine returns an error message, could imply a bug. We manually examined some of those tests. This oracle is particularly useful with ASTGen generators which can intentionally create programs that do not meet the preconditions.

- **Different outputs oracle:**

  This oracle is used in differential testing [7]. It takes an input program and a refactoring and feeds this pair to both NetBeans and Eclipse. It then takes the refactored programs returned by the two engines and checks whether their ASTs are equivalent using the AST Comparator (Section 1.1). If the two ASTs differ, we manually inspect the two refactored programs to check whether the difference is caused by a bug in one of the refactoring engines. When the refactored programs are equivalent, they are not necessarily correct, as both engines can have the same bug.

- **Different behavior oracle:**

  This last oracle is used to check that the refactoring preserves the original behavior at run time.

It runs an input program and its corresponding refactored program, and compares their text outputs. We only use this oracle when the input programs in refactoring tests include a main method.

Table 2.7 shows the oracles used for each group of refactoring tests that can result after performing the refactorings in NetBeans and Eclipse. The first column lists the group of tests, and the second column shows the oracles used for that group of tests.

Group of tests	**Oracles**
NONE	n/a
NB	Erroneously refactored Do not compile Not refactored Different behavior
EC	Do not compile Not refactored Different behavior
BOTH	Do not compile Different outputs Different behavior

Table 2.7: Oracles used per group of tests

2.6 Procedure to find bugs

This section describes the procedure used to find bugs for both Eclipse tests and ASTGen tests. For each test suite, we present the following information: description about the test suite and the refactoring it tests, summary table for the results, detailed analysis of the results per oracle, and a list of the reported bugs.

- **Description:**

  This part provides information about the refactoring associated to this test suite. It includes a list of the characteristics for the tests such as the refactoring parameters and some variations among the tests. It finally shows two tests from this test suite.

- **Summary table:**

  Table 2.8 shows a general summary table that includes all oracles. The first column shows the group of tests, the second column lists the number of tests, the third column shows the number of tests that crash, the fourth column shows the number of tests Erroneously refactored, the fifth column shows the number of tests Not refactored, the sixth column shows the number of tests that Do not compile, the seventh column shows the number of tests that have different outputs, and the last column shows the number of tests that result in change of behavior. We use the following acronyms: EP=Erroneously refactored, NR=Not refactored, DNC=Do not compile, Diff=Different outputs, and COB=Different behavior. In the summary table, we only include those oracles that have a total number of tests greater than zero.

		Oracles					
Group of tests	**# Tests**	**Crash**	**EP**	**NR**	**DNC**	**Diff**	**COB**
NONE							
NB							
EC							
BOTH							
Totals							

Table 2.8: Summary table for refactoring tests

- **Detailed results for each oracle:**

  We include a detailed explanation for each oracle included in the summary table. Depending on the oracle, it will include a list of different error messages, different compilation errors, differences between output files, or different changes of behavior.

- **List of bugs found:**

  We list the bugs that were found using the oracles and show two of these bugs. We manually identify the bugs, taking into account that several compilation errors could be related to the same bug. The format used to describe the bugs includes the following:

  - **Refactoring:** This text describes the refactoring that will be performed. It includes all the refactoring parameters.

- **Problem:** This text explains the main problem for this bug. It can be a compilation error or a change in behavior.
- **Solution:** This text explains a suggested solution for this bug.
- **Note:** This text provides other possible scenarios for this bug. This is an optional information for a bug.
- **Before:** This text shows the input program. It includes all the input program's input files.
- **After:** This text shows the refactored program. It includes the relevant output files to understand the bug.

Chapter 3

Using Eclipse tests

This chapter presents our results from running some existing Eclipse refactoring tests on NetBeans. It first summarizes the existing Eclipse tests (Section 3.1). It then explains the differences between NetBeans and Eclipse refactorings (Section 3.2). It next lists the groups of Eclipse tests (Section 3.3). It also presents a specific example using Eclipse tests (Section 3.4). It finally presents a summary of our results (Section 3.5) and some discussion related to this type of tests (Section 3.6).

This chapter is focused on reusing existing Eclipse tests in NetBeans and finding bugs using differential testing (together with several other oracles). We first selected 5 refactorings to test. We then wrote runners for their corresponding Eclipse tests. After performing the refactorings in NetBeans using our refactoring runners, we evaluated the differences between Eclipse and NetBeans output files and reported new bugs in NetBeans.

There are some reasons to believe that the Eclipse refactoring engine is much more stable than NetBeans refactoring engine. First, the Eclipse refactoring engine was started three years before the NetBeans refactoring engine, and during that time the Eclipse refactoring engine was used and tested by many users who reported the bugs they found to Eclipse Bugzilla, the bug reporting system for Eclipse projects. Second, Eclipse has more refactoring tests than NetBeans. Since NetBeans refactoring engine is a relatively young tool, we expected that reusing Eclipse tests could find similar bugs in NetBeans to those previously reported to Eclipse.

3.1 List of Eclipse tests per refactoring

This section provides a list of Eclipse tests for each refactoring included in Eclipse version 3.2. Some of these tests were written as the refactorings were developed, and some new tests were added based on the bugs reported to Eclipse Bugzilla. To the best of our knowledge, all tests were written manually. They can be found in the package `org.eclipse.jdt.ui.tests.refactoring` (available from the CVS repository for Eclipse source but not included in the default Eclipse source distributions). The refactorings Rename and Move have more than one test suite, and for that reason they are presented in separated tables.

3.1.1 Rename

This refactoring can be performed over several program elements such as methods, fields, and types. Table 3.1 shows the list of Eclipse tests for Rename. The first column lists the program elements, the second column lists the test suites for the program element, and the third column lists the number of tests contained in a test suite. We did not find test suites for these program elements: Enum Constant and Compilation Unit.

Program Element	**Test Suite**	**# Tests**
Method	RenameMethodInInterfaceTests	95
	RenameVirtualMethodInClassTests	88
	RenamePrivateMethodTests	23
	RenameStaticMethodTests	27
	RippleMethodFinderTests	12
Parameter	RenameParametersTests	58
Field	RenameNonPrivateFieldTests	52
	RenamePrivateFieldTests	23
Local Variable	RenameTempTests	81
	RenamingNameSuggestorTests	8
Type	RenameTypeTests	221
Type Parameter	RenameTypeParameterTests	21
Enum Constant	-	-
Compilation Unit	-	-
Package	RenamePackagesTests	34
Source Folder	RenameResourceChangeTests	6
	RenameSourceFolderChangeTests	3
Project	RenameJavaProjectTests	2
Total		**754**

Table 3.1: Eclipse tests for Rename

Program Element	Test Suite	# Tests
Instance method	MoveInstanceMethod	50
Type Method Field	MoveMembers	89
Total		**139**

Table 3.2: Eclipse tests for Move

3.1.2 Move

This refactoring can be performed over several program elements such as instance methods, inner classes, and fields. Table 3.2 shows the list of Eclipse tests for Move. The first column lists the program elements, the second column lists the test suites, and the third column lists the number of tests contained in a test suite.

3.1.3 Remaining refactorings

Table 3.3 includes the remaining core refactorings. The first column shows the refactoring name, the second column lists the test suite for this refactoring, and the last column shows the number of tests for this test suite.

Refactoring	Test Suite	# Tests
ExtractSuperclass	ExtractSuperTypeTests	5
ExtractInterface	ExtractInterfaceTests	126
UseSupertypeWherePossible	UseSuperTypeWherePossibleTests	110
PushDown	PushDownTests	95
PullUp	PullUpTests	139
ChangeMethodSignature	ChangeSignatureTests	127
ExtractMethod	ExtractMethodTests	389
ExtractLocalVariable	ExtractTempTests	127
ExtractConstant	ExtractConstantTests	58
Inline	InlineConstantTests InlineMethodTests InlineTempTests	32 163 45
ConvertAnonymousClassToNested	ConvertAnonymousToNestedTests	43
ConvertMemberTypeToTopLevel	MoveInnerToTopLevel	84
ConvertLocalVariableToField	PromoteTempToFieldTests	52
IntroduceIndirection	IntroduceIndirectionTests	31
IntroduceFactory	IntroduceFactoryTests	44
IntroduceParameter	IntroduceParameterTest	20
EncapsulateField	SefTests	32
GeneralizeDeclaredType	ChangeTypeRefactoringTests	58
InferGenericTypeArguments	InferTypeArgumentsTests	58
Total		**1838**

Table 3.3: Eclipse tests for Remaining Refactorings

3.2 Differences between NetBeans and Eclipse refactorings

Ideally, applying the same refactoring to a program should produce the same result independently of the IDE refactoring engine. In practice, this is not true because there can be some differences related to different IDE's refactoring interpretations or bugs found in the refactoring engines.

When we perform a refactoring, there is a number of options that we can specify. These options vary depending on the refactoring engine. Table 3.4 shows the possible options in NetBeans and Eclipse for the selected refactorings. The first column lists the refactorings, the second column lists all available options in NetBeans for this refactoring, and the third column lists all available options in Eclipse.

For example, while PushDownMethod in Eclipse has two options— `Leave an abstract declaration` and `Add required`— in NetBeans it has only one option— `Keep an abstract declaration`. Eclipse in general provides more options for each refactoring. This difference implies that there will be some Eclipse tests that cannot be performed in NetBeans.

3.3 Groups of Eclipse tests

We inspected Eclipse tests and found that they can be divided into these five groups.

- **Tests that are expected to return an error message:**

  These are tests for which NetBeans returns an error message because some preconditions are not satisfied to apply the refactoring. We can identify these tests by their name. Some of these tests follow this pattern: `testFail*` or `test*Fail*` but there are also other tests with names such as `testWrongArg*` or `testIllegalInnerClass`.

- **Tests that validate that a refactoring is available:**

  These tests only check that a refactoring can be performed in some scenarios, but they do not validate whether the result was correct or incorrect. We can identify these tests by their name.

Refactoring	NetBeans's options	Eclipse's options
RenameClass	• Apply Rename on comments	• Update references • Update similarly named variables and methods • Update textual occurrences on comments and strings • Update fully qualified names in non-Java text files
RenameMethod	• Apply Rename on comments	• Update references • Keep original method as delegate to Rename method • Mark as deprecated
RenameField	• Apply Rename on comments	• Update references • Update textual occurrences on comments and strings • Rename getter method • Rename setter method
MoveType	n/a	n/a
PullUpMethod	• Make abstract	• Declare abstract in destination • Add required • Use destination type where possible • Use destination type in `instanceOf` expressions
PullUpField	n/a	• Add required • Use destination type where possible • Use destination type in `instanceOf` expressions
PushDownMethod	• Keep an abstract declaration	• Leave an abstract declaration • Add required
PushDownField	n/a	• Add required
EncapsulateField	• Create getter/setter • Field's visibility • Accessor's visibility • Use accessors even when field is accessible	• Insert new methods after • Access modifier • Field access in declaring type • Generate method comments

Table 3.4: Refactoring's options in NetBeans and Eclipse

For instance, in the case of the PushDown refactoring we have the following pattern for these tests: `testEnablement*`.

- **Tests that validate that refactoring options works correctly:**
  These tests only check that the refactoring options works correctly. We can identify these tests by their name. For instance, in the case of the PullUp refactoring we have the following pattern: `testAddingRequiredMembers*`. One of these tests could validate that the optional parameter `Add required` from the PushDownMethod refactoring adds the required program elements to perform the refactoring.

- **Tests related to common functionality between NetBeans and Eclipse:**
  These are tests that validate that the refactoring produce a refactored program for the common scenarios between NetBeans and Eclipse.

- **Tests related to different functionality between NetBeans and Eclipse:**
  These tests check that the refactoring produce a refactored program even when additional options in Eclipse are selected. For instance, one of these tests can validate that the PushDownMethod refactoring with optional parameter `Add required` produce a refactored program.

From these five groups of tests, we selected the first and the fourth groups of tests because they can be applied to both NetBeans and Eclipse. These groups of tests allow us to use differential testing (see Section 1.1) to find bugs in NetBeans.

3.4 Example

We tested five refactorings in NetBeans using Eclipse tests (Section 2.4). This section explains one of these refactorings and lists the bugs that were found using Eclipse tests for this refactoring.

3.4.1 PullUp

This refactoring can move one or more fields and methods from an origin class to any of its existing superclasses in the class hierarchy. The PullUp refactoring takes as input the selected program elements and the superclass. When we apply the PullUpMethod refactoring, we can additionally

specify to make an abstract declaration in the superclass using the option `Make Abstract`. The PullUp refactoring without setting `Make Abstract` performs the following transformations:

- Removes the selected methods and fields in the origin class.
- Removes methods implementation and fields corresponding to the selected methods and fields in all direct subclasses of the superclass.
- Creates the selected methods and fields in the superclass.

The PullUp refactoring setting `Make Abstract` performs this transformation:

- An abstract method that corresponds to the selected method is added to the superclass.

The PullUp refactoring checks several preconditions, including that the selected methods or fields do not already exist in the superclass and that the selected methods do not refer to a field or method not accessible in the superclass. Table 3.5 shows a sample program before and after PullUpMethod B.m() to superclass A setting 'Make Abstract'.

Before:

```
A.java
class A{
}
class B extends A{
    void m(){
    }
}
```

After:

```
A.java
abstract class A{
    protected abstract void m();
}
class B extends A{
    protected void m(){
    }
}
```

Table 3.5: Example for PullUp setting 'Make Abstract'

3.4.2 List of bugs found using Eclipse tests

We found 6 bugs for the PullUp refactoring in NetBeans using Eclipse tests; 1 bug is related to PullUpField, and the other 5 bugs are related to PullUpMethod. They correspond to bugs 111561, 111563, 111564, 111565, 111566, and 111567. Table 3.6 shows bug 111565.

Bug:	#111565
Refactoring:	PullUpMethod B.m() and set 'Make Abstract'
Problem:	Compilation error: B1.m() cannot override A.m(); attempting to assign weaker access privileges; was public
Solution:	It should return an error message such as: "Method B1.m() has visibility lower than default, which will result in compilation errors if you proceed"

Before:

```
A.java
class A{
}
class B extends A{
  public void m() {
  }
}
class B1 extends A{
  private void m() {
  }
}
```

After:

```
A.java
abstract class A{
  public abstract void m();
}
class B extends A{
  public void m() {
  }
}
class B1 extends A{
  private void m() {
  }
}
```

Table 3.6: Bug #111565 for PullUpMethod

3.5 Results using Eclipse tests

This section first includes the summary of our results using Eclipse tests. It then explains the results for 3 relevant Eclipse tests.

3.5.1 Summary of results using Eclipse tests

Table 3.7 shows the list of Eclipse tests that were run in NetBeans version 6.0 M3. The first column shows the refactoring, the second column shows the test suite, the third column shows the number of tests for the test suite, the fourth column shows the number of tests that was run in NetBeans, and the next three columns show these oracles: Not refactored, Do not compile, and Different outputs. The last column shows the number of reported bugs. Some test suites are not applicable for NetBeans, and the symbol `n/a` is used to indicate this. We omitted the word 'Tests' at the end of each test suite.

				Oracles			
Refactoring	**Test suite**	**# of tests**	**# of run tests**	**NR**	**DNC**	**Diff**	**Bugs reported**
Rename (Method)	RenameMethodInInterface	97	97	27	25	13	4
Rename (Method)	RenameVirtualMethodInClass	88	69	19	14	46	1
Rename (Method)	RenamePrivateMethod	23	22	1	1	9	1
Rename (Method)	RenameStaticMethod	27	21	6	6	12	1
Rename (Field)	RenamePrivateField	23	16	7	1	16	0
Rename (Field)	RenameNonPrivateField	52	42	10	17	31	1
Rename (Type)	RenameType	227	181	24	37	8	8
ConvertMemberTypeToTopLevel	MoveInnerToTopLevel	87	54	9	21	16	3
PushDown(Method) PushDown(Field)	PushDown	95	64	5	17	40	0
PullUp(Method) PullUp(Field) PullUp(Member type)	PullUp	139	65	39	22	72	6
EncapsulateField	SefTests	28	26	2	9	15	2
						Total Bugs:	**27**

Table 3.7: Summary of Eclipse tests in NetBeans

3.5.2 Detailed results for relevant Eclipse tests

We provide explanation of our results for the following Eclipse tests: RenameMethodInInterface, RenameType, and MoveInnerToTopLevel.

Test suite: RenameMethodInInterface

The common case for these tests involve to Rename method I.m() to k, where I is an interface. Optionally, any class or interface in the hierarchy implements I and declares method k. These are five variations for these tests:

- visibility of methods m and k
- method m can be native, have different parameters, and have different return types
- methods m and k are declared in different interfaces
- method k is declared in a class that optionally implements one or more interfaces
- method k can be a special or forbidden method such as wait, main, toString, hashCode, notify, or notifyAll

Two examples:

We show two tests of RenameMethodInInterface. Figure 3.1 shows a simple test, and Figure 3.2 shows a more complex test.

test2, Rename method I.m to k

```
package p;
class A implements I{
        public void m(){};
        public void m(int y){};
}
interface I {
        void m();
}
```

Figure 3.1: test2 for RenameMethodInInterface

test30, Rename method I.m to k

```
package p;
interface I{
        void m();
}
class A implements I, J{
        public void m(){};
}
interface J{
        void m();
}
class B implements J, K{
        public void m(){};
}
interface K{
        void m();
}
```

Figure 3.2: test30 for RenameMethodInInterface

Explanation of the results:

This test suite has 97 tests. For 70 tests NetBeans produced a refactored program, and for 27 tests NetBeans returned an error message. Table 3.8 shows the summary results for this test suite.

		Oracles			
Group of tests	**# Tests**	**NR**	**EP**	**DNC**	**Diff**
NONE	21	21	n/a	n/a	n/a
EC	6	6	n/a	n/a	n/a
NB	24	n/a	24	11	n/a
BOTH	46	n/a	n/a	14	13
Totals	97	27	24	25	13

Table 3.8: Summary for RenameMethodInInterface

- **Not refactored**

  There are 27 tests for which NetBeans returned an error message. 6 tests included Java 1.5 features, and 21 were tests expected to return an error message. These are the different error messages that were returned by NetBeans:

  ```
  - Cannot rename.  Method k with the same signature already exists in class A
  - After renaming, the original method A.m() will be overridden by method k in B with
    weaker access privileges
  - This method overrides or implements methods in super classes or interfaces, so its
    name cannot be changed
  - After renaming, method I.m will override the original final method getClass in
    java.lang.Object
  - Can't find resource for bundle org.openide.util.NbBundle$PBundle, key LBL_instance
  ```

  Tests that returned the last error message were manually performed in NetBeans version 6.0 M9 and returned this compilation error:

  ```
  - C2.k() cannot override C1.k(); overridden method is static
  ```

- **Erroneously refactored**

  There are 24 tests that were erroneously refactored in NetBeans. They should have returned the following error messages:

  ```
  - Related method 'm' is native.  Renaming will cause an UnsatisfiedLinkError at runtime
  - A related type declares a method with the new name (and same number of parameters)
  ```

- **Do not compile**

  There are 25 tests for which NetBeans produced a refactored program that did not compile. 11 of them correspond to tests that were expected to return an error message. These are their compilation errors:

  ```
  - B.k() cannot implement I.k(); attempting to assign weaker access privileges; was public
  ```

```
- I.toString() cannot override toString() in java.lang.Object; attempting to use incompa
  tible return type
```

The other 14 tests correspond to tests that were expected to generate a refactored program. This is the compilation error for these tests:

```
- C is not abstract and does not override abstract method m() in J
```

- **Different outputs**

  There are 13 tests for which NetBeans and Eclipse produced refactored programs with different output files. The difference for these tests is related to:

```
- There is no method corresponding to k
```

  NetBeans did not rename all related methods m to k.

Bugs found using this test:

We found 4 bugs for Rename using RenameMethodInInterface. They correspond to bugs 99419, 99481, 99596, and 111593. Table 3.9 shows bug 99596, and Table 3.10 shows bug 111953.

Bug:	#99596
Refactoring:	RenameMethod I.m() to k
Problem:	Compilation error: k() is already defined in C
Solution:	It should return a message such as: "A related type declares a method with the new name (and same signature)"

Before:

```
C.java
interface I{
  void m();
}
class C implements I{
  public void m(){}
  public void k(){}
}
```

After:

```
C.java
interface I{
  void k();
}
class C implements I{
  public void k(){}
  public void k(){}
}
```

Table 3.9: Bug #99596 for RenameMethod

Bug:	#111953
Refactoring:	RenameMethod I.m() to k
Problem:	Change of behavior: the refactoring did not RenameMethod B.m to k
Solution:	Method B.m() should be renamed to k

Before:

```
A.java
class B {
  public void m(){};
}
class A extends B implements I{
  public void m(){};
}
interface I {
  void m();
}
```

After:

```
A.java
class B {
  public void m(){};
}
class A extends B implements I{
  public void k(){};
}
interface I {
  void k();
}
```

Table 3.10: Bug #111953 for RenameMethod

Test suite: RenameType

The common case for these tests involve to RenameType A to B. Optionally, the type B could exist in the hierarchy. These are two variations for these tests:

- Type A can be referenced by a field, a constructor, a method parameter, a method return type, or a local variable in type A or any class in its hierarchy.
- Type A can be referenced by a class that is not in its hierarchy.

Two examples:

We show two tests of RenameType. Figure 3.3 shows a simple test, and Figure 3.4 shows a more complex test.

test5, Rename type A to B

```
class A{
   void m(A a){
   };
}
```

Figure 3.3: test5 for RenameType

test50, Rename type A to B

```
class A{
  A( ){};
};
class C extends A{
  C(){
    super();
  }
}
```

Figure 3.4: test50 for RenameType

Explanation of the results:

This test suite has 227 tests, of which 181 tests were refactored in NetBeans. For 157 tests NetBeans produced a refactored program, and for 24 tests NetBeans returned an error message. 46 tests were discarded because they are related to additional functionality in Eclipse. Table 3.11 shows the summary results for this test suite.

		Oracles			
Group of tests	**# Tests**	**NR**	**EP**	**DNC**	**Diff**
NONE	23	23	n/a	n/a	n/a
EC	1	1	n/a	n/a	n/a
NB	91	n/a	91	30	n/a
BOTH	66	n/a	n/a	7	8
	181	24	91	37	8

Table 3.11: Summary for RenameType

- **Not refactored:**

  There are 24 tests for which NetBeans returned an error message. 23 were tests expected to return an error message, and 1 test could not be run using our environment. These are the different error messages that were returned by NetBeans:

  `- Cannot rename.  Inner class B already exists in class Outer`

  `- Cannot rename.  Class B already exists in this folder`

  `- ‘‘this’’ is not a valid Java identifier`

- **Erroneously refactored:**

  There are 91 tests that were erroneously refactored in NetBeans. They should have returned the following error messages:

  - `After the refactoring the method length will reference a local variable instead of type A.`
  - `Another type named B is referenced in A`
  - `method j in type A.I.C is native. Running the modified program will cause UnsatisfiedLinkError.`

- **Do not compile:**

  There are 37 tests for which NetBeans produced a refactored program that did not compile. 30 of them correspond to tests that were expected to return an error message. These are their compilation errors:

  - `duplicate class: B`
  - `B is already defined in p`
  - `cyclic inheritance involving p.B`
  - `cannot assign a value to final variable length`
  - `error while writing p.aux: (The system cannot find the file specified)`

  The other 7 tests correspond to tests that were expected to generate a refactored program. These are their compilation errors:

  - `package p.A does not exist`
  - `'(' or '[' expected`

- **Different outputs:**

  There are 8 tests for which NetBeans and Eclipse produced a refactored program with different output files. The difference for these tests is related to:

  - `fields String aa = 'C:\B.java'; and String aa = 'C:\A.java'; are not equal`

    NetBeans does not update a reference to type A inside a String.

Bugs found using this test:

We found 8 bugs for Rename using RenameType. They correspond to bugs 100297, 100298, 100299, 100300, 100302, 111965, 111966, and 111967. Table 3.12 shows bug 100300, and Table 3.13 shows bug 100302.

Bug:	#100300
Refactoring:	RenameType A to B
Problem:	Compilation error: package p1.B clashes with class of same name
Solution:	It should return a message such as: "Type name B will collide with a package name"

Before:

```
A.java
package p1;
class A{
}

C.java
package p1.B;
class C{
}
```

After:

```
B.java
package p1;
class B {
}

C.java
package p1.B;
class C {
}
```

Table 3.12: Bug #100300 for RenameType

Bug:	#100302
Refactoring:	RenameType A to B
Problem:	Change of behavior
Solution:	It should return a message explaining that the refactoring will result in change of behavior

Before:

```
A.java
class A{
   static int length = 17;
   int m(int[] B){
     return A.length;
   }
}
```

After:

```
B.java
class B{
   static int length = 17;
   int m(int[] B){
     return B.length;
   }
}
```

Table 3.13: Bug #100302 for RenameType

Test suite: MoveInnerToTopLevel

The common case for these tests involve to MoveInnerClassToOuterLevel. Optionally a field can be declared for the current outer class. These are two variations for these tests:

- Class Inner can reference a field or a method of an outer class.
- Class Inner can be referenced by a field or a method in an outer class.

Two examples:

We show two tests of MoveInnerToTopLevel. Figure 3.5 shows test10, and Figure 3.6 shows test25.

test10, Move class Inner to outer level

```
class A {
  static int F= 1;
  static class Inner{
    void foo() {
      F= 2;
    }
  }
}
```

Figure 3.5: test10 for MoveInnerToTopLevel

test25, Move class Inner to outer level

```
class A {
  public static class Inner {
  }
  public A(){
    super();
    new A.Inner();
  }
}
```

Figure 3.6: test25 for MoveInnerToTopLevel

Explanation of the results:

This test suite has 87 tests, of which 54 tests were refactored in NetBeans. For 45 tests NetBeans produced a refactored program, and for 9 tests NetBeans returned an error message. 33 tests were discarded because they are related to additional functionality in Eclipse (for instance, Eclipse moves

inner and local classes, while NetBeans only moves inner classes). Table 3.14 shows the summary results for this test suite.

		Oracles			
Group of tests	**# Tests**	**NR**	**EP**	**DNC**	**Diff**
NONE	2	2	n/a	n/a	n/a
EC	7	7	n/a	n/a	n/a
NB	2	n/a	2	2	n/a
BOTH	43	n/a	n/a	19	16
Totals	54	9	2	21	16

Table 3.14: Summary for MoveInnerToTopLevel

- **Not refactored:**

  There are 9 tests for which NetBeans returned an error message. 2 of them were not included by Eclipse, and the other 7 were tests expected to generate a refactored program, but that return an error message. These are the different error messages that were returned by NetBeans:

  ```
  - Field named <b>a</b> already exists in this class
  ```

  ```
  - Class A.Inner not found
  ```

  Tests that returned the last error message were manually performed in NetBeans version 6.0 M9 and returned this compilation error:

  ```
  - cannot find symbol:  variable E; cannot find symbol:  method foo()
  ```

- **Erroneously refactored:**

  There are 2 tests that were erroneously refactored in NetBeans. They should have returned the following error messages:

  ```
  - Name 'a' is used as a parameter name in one of the constructors of type Inner
  ```

  ```
  - Type named Inner already exists in package p
  ```

- **Do not compile:**

  There are 21 tests for which NetBeans produced a refactored program that did not compile. 2

of them correspond to tests that were expected to return an error message. These are their compilation errors:

- `a is already defined in Inner`
- `duplicate class:  Inner`

The other 19 tests correspond to tests that were expected to generate a refactored program. They correspond to the following compilation errors:

- `cannot find symbol:  variable F`
- `package Inner2 does not exist`
- `<identifier expected>`
- `modifier protected not allowed here`
- `a has private access in SomeClass`

- **Different outputs:**

  There are 16 tests for which NetBeans and Eclipse produced refactored programs with different output files. They correspond to the following differences:

  - `type A, fields Inner inner = new Inner(); and innertotop.MoveInnerToTopLevel.test23.in. Inner inner = new innertotop.  MoveInnerToTopLevel.test23.in.Inner(); are not equal`

    These are false positives related to the way NetBeans input files are loaded, e.g., `package innertotop.MoveInnerToTopLevel.test23.in.A.java` in NetBeans versus `package p` in Eclipse.

  - `type Inner, fields /** Comment */ private final A a; and private final A a; are not equal`

    These are false positives related to our AST Comparator that does not omit comments.

Bugs found using this test:

We found 3 bugs using MoveInnerToTopLevel test suite. They correspond to bugs 100305, 100306, and 100308. Table 3.15 shows bug 100305, and Table 3.16 shows bug 100308.

Bug: #100305

Refactoring: MoveClassInnerToOuterLevel declaring field "a" for the outer class

Problem: Compilation error: a is already defined in Inner

Solution: It should return a message such as: "Name 'a' is used as a parameter name in one of the constructors of type Inner"

Before:

```
A.java
class A{
  class Inner{
    Inner(int a){
    }
  }
}
```

After:

```
Inner.java
class Inner{
    private final A a;
    Inner(A a, int a){
        this.a = a;
    }
}
```

Table 3.15: Bug #100305 for MoveClassInnerToOuterLevel

Bug: #100308

Refactoring: MoveClassInnerToOuterLevel declaring field 'a' for the current outer class

Problem: Compilation error: cannot find symbol: variable E

Solution: Class Inner should include the import statement

Before:

```
A.java
import static java.lang.Math.E;
public class A {
  class Inner {
    public void doit() {
      foo();
      double e= E;
    }
  }
  static void foo(){};
}
```

After:

```
Inner.java
class Inner {
    public Inner(A a) {
        this.a = a;
    }
    private final A a;
    public void doit() {
        this.a.foo();
        double e = E;
    }
}
```

Table 3.16: Bug #100308 for MoveClassInnerToOuterLevel

3.6 Discussion

One complication to run Eclipse tests in NetBeans was that Eclipse uses several structures to store the tests. Having only one structure would make it easier to automate these tests not only for NetBeans but also for Eclipse.

We found that NetBeans and Eclipse have different preconditions for some refactorings. There were some Eclipse tests that were expected to returned an error message for which NetBeans produced a refactored program. For example, the EncapsulateField refactoring returns an error message in Eclipse and a refactored program in NetBeans when the input program has a getter or setter.

Chapter 4

Using ASTGen tests

This chapter presents our results from running automatically generated tests using ASTGen [6] on NetBeans. It first describes some features of ASTGen (Section 4.1). It next summarizes our test generators (Section 4.2). It then presents a specific example using ASTGen tests (Section 4.3). It finally presents a summary of our results (Section 4.4) and some discussion related to this type of tests (Section 4.5).

Writing refactoring tests manually is tedious, expensive, and results in incomplete test suites, potentially leaving many hidden bugs in refactoring engines. We used ASTGen generation (see Section 1.1) to automate testing of refactoring engines. We first selected 6 refactorings to test. We then wrote several ASTGen generators which automatically produced refactoring tests. We next wrote a runner for each test generator. We finally ran the generated tests in NetBeans version 6.0 M3 and Eclipse version 3.2, and found bugs in both NetBeans and Eclipse using differential testing and several other oracles [6].

4.1 ASTGen features

ASTGen is a framework that offers a library of generic, reusable, and composable generators that produce abstract syntax trees (ASTs). Using ASTGen, a developer can focus on the creative aspects of testing rather than the mechanical production of test inputs. Instead of manually writing input programs, a developer writes a generator whose execution produces thousands of programs with structural properties that are relevant for the specific refactoring being tested. Figure 4.1 shows some outputs for three basic generators, and Figure 4.2 shows two outputs for 'DoubleClassFieldRefGenerator', which is a test generator that uses the previous basic generators.

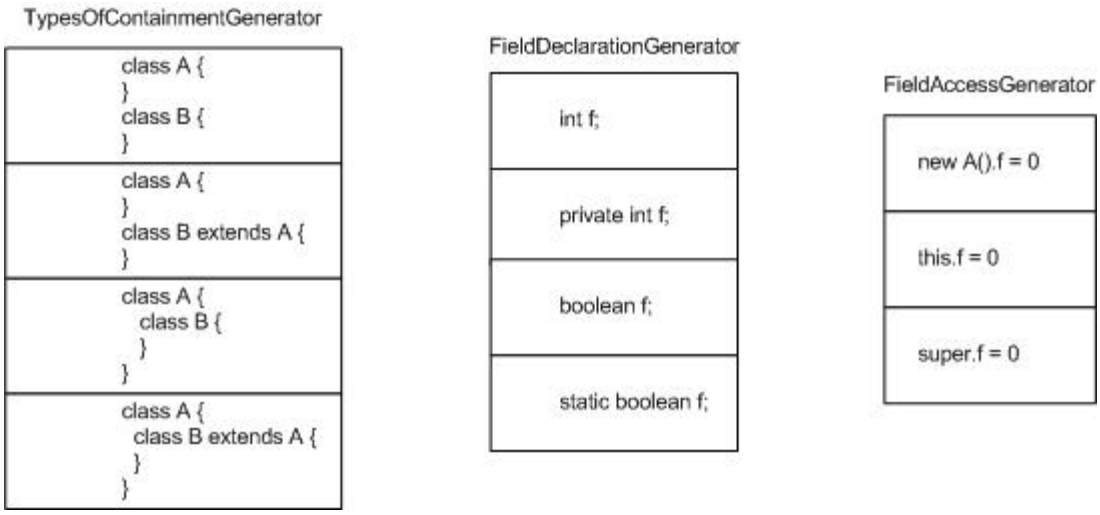

Figure 4.1: Three basic generators

```
class A {
  int f;
  class B {
    void m() {
      (new A().f) = 0;
    }
  }
}
```

Figure 4.2: One composed generator

ASTGen follows the *bounded-exhaustive* approach [10, 11, 12, 13, 14] for exhaustively testing all inputs within the given bound. This approach covers all "corner cases" within the given bound. In contrast, manual testing requires identifying each corner case and covering it with a manually written test. Bounded-exhaustive testing has not been used before for test inputs as complex as Java programs, and the approach of imperative generators used by ASTGen differs from the previous techniques using declarative generators [10, 11, 12, 13, 14]. More comparison with related work is discussed in Chapter 5.

4.2 List of ASTGen generators

We use several ASTGen generators to generate input programs for the selected refactorings. These generators reuse basic generators from the ASTGen library [6] to create our specific generators.

For instance, the 'DoubleClassFieldReference' is created using these 3 basic generators: 'AllPossibleRelationships', 'FieldDeclaration', and 'FieldReferences'. The last 8 test generators were written by the author of this thesis:

- **ClassArrayField:** Generates classes, each of which declares a field of some array type.
- **ClassRelationship:** Generates pairs of classes that are related in many ways.
- **FieldReference:** Generates many classes. Each class contains a field and a method that references the field in many ways.
- **DoubleClassFieldReference:** Generates pairs of classes related in many ways. One class declares a field and the other references it in many ways.
- **MethodReference:** Generates many classes with two methods. One method calls the other and may overload it.
- **MethodParamReference:** Generates method declarations, each of which has a parameter referenced.
- **SingleClassTwoFields:** Generates classes, each of which declares two fields. One field references the other in its initialization and the main method prints the values for both fields.
- **DoubleClassGetterSetter:** Generates pairs of classes in an inheritance relationship. The superclass declares a field, either the superclass or subclass declares a getter or setter for the field, and the superclass declares a main method that references the getter or setter.
- **DoubleClassParentField:** Generates pairs of classes in an inheritance relationship. The superclass declares a field and either the superclass or subclass declares another field that references the first one its initializer.
- **DoubleClassParentMethod:** Generates pairs of classes in an inheritance relationship. The superclass declares a field, the subclass declares a method that optionally references the field, and either the superclass or subclass declares a second method that references the first method.

- **DoubleClassChildField:** Generates pairs of classes in an inheritance relationship. The superclass declares a field, and the subclass declares another field that optionally references the first one in its initializer.
- **DoubleClassChildMethod:** Generates pairs of classes in an inheritance relationship. The superclass declares a field, the the subclass declares two methods, one of these methods optionally references the field, and the other method references the first method.
- **TripleClassChildField:** Generates triples of classes such that two of them have an inheritance relationship. The subclass declares a field that references another field declared in either of the other classes.
- **TripleClassChildMethod:** Generates triples of classes such that two of them have an inheritance relationship. The subclass declares two methods. The first method references a field declared in one of the other two classes, and the second method references the first method.

4.3 Example

We tested 6 refactorings in NetBeans using the generated tests (Section 2.4). This section explains one of these refactorings and lists the bugs found using the generated tests for this refactoring.

4.3.1 EncapsulateField

This refactoring replaces all references to a field with accesses through setter and getter methods. It takes as input the name of the field to encapsulate and the names of the new getter and setter methods. It performs the following transformations:

- creates a public getter method that returns the field's value
- creates a public setter method that updates the field's value to a given parameter's value
- replaces all field reads with calls to the getter method
- replaces all field writes with calls to the setter method
- changes the field's access modifier to private.

The EncapsulateField refactoring checks several preconditions, including that the code does not already contain accessor methods and that these methods are applicable to the expressions in which the field appears. Table 4.1 shows a sample program before and after EncapsulateField A.f.

Before:

A.java

```
class A {
  public int f;
  void m(int i) {
    f = i * f;
  }
}
```

After:

A.java

```
class A {
  private int f;
  void m(int i) {
    setF(i * getF());
  }
  public int getF() {
    return this.f;
  }
  public void setF(int f) {
    this.f = f;
  }
}
```

Table 4.1: Example for EncapsulateField

4.3.2 List of bugs found using ASTGen tests:

We found 8 bugs for EncapsulateField using 5 ASTGen generators. They correspond to bugs 98054, 98055, 98056, 98057, 98165, 108473, 108474, and 108489. Table 4.2 shows bug 99481.

Bug:	#99481
Refactoring:	EncapsulateField A.b
Problem:	Compilation error: setB(byte) cannot be applied to (int)
Solution:	It is necessary to add casting from int to byte for 'getB() + 1'.

Before:

A.java

```
class A {
  byte b;

  void m() {
    b++;
  }
}
```

After:

A.java

```
class A {
  private byte b;
  void m() {
    setB(getB() + 1);
  }
  public byte getB() {
    return b;
  }
  public void setB(byte b) {
    this.b = b;
  }
}
```

Table 4.2: Bug #99481 for RenameMethod

4.4 Results using ASTGen tests

This section first includes the summary of our results using ASTGen tests. It then explains in detail the results for 3 relevant ASTGen tests.

4.4.1 Summary of results using ASTGen tests

Table 4.3 shows the list of ASTGen test generators that were run in NetBeans. The first column lists the refactorings. The second column lists the test generators. The next three columns show information related to the tests generation such as the total number of generated tests, the total generation time, and the number of compilable inputs. The next six columns show information related to oracles such as number of tests that return an error message, number of tests that do not compile after having performed the refactoring, number of tests which invertible programs are correct, and number of tests that have different outputs in NetBeans and Eclipse. The last two columns show the number of bugs reported in Eclipse and NetBeans. We use the following acronyms: CS = ChangeMethodSignature, Ecl = Eclipse, NB = NetBeans, Generation: TGI = Total Generated Inputs, Time is in min:sec, CI = Compilable Inputs; Oracles: NR = Not refactored, DNC = DoesNotCompile, C/I = Custom/Inverse, Diff. = Differential.

Refactoring	Generation				Oracles						**Bugs Reported**	
	Test generator	TGI	Time	CI	NR		DNC		C/I	Diff.		
					Ecl	NB	Ecl	NB			Ecl	NB
Rename Class)	ClassRelationships	108	1:02	88	0	0	0	0	0	0	0	0
Rename (Method)	MethodReference	9540	89:12	9540	0	0	0	0	0	0	0	0
Rename (Field)	FieldReference	3960	28:20	1512	0	0	0	304	0	40	0	1
Rename (Field)	DoubleClassFieldRef.	14850	76:55	3969	0	0	0	0	0	0	0	0
EncapsulateField	ClassArrayField	72	0:45	72	0	0	48	0	0	48	1	0
	FieldReference	3960	15:19	1512	0	0	320	432	14	121	4	3
	DoubleClassFieldRef.	14850	41:45	3969	0	0	187	256	100	511	1	2
	SingleClassTwoFields	60	1:16	48	0	0	0	0	48	15	1	0
	DoubleClassGetterSetter	576	8:45	417	216	0	162	162	18	0	3	3
PushDownField	DoubleClassFieldRef.	4635	10:56	1064	760	380	152	228	0	380	2	3
	DoubleClassParentField	360	6:50	270	246	168	18	90	0	78	1	1
PushDownMethod	DoubleClassParentMethod	960	17:11	820	784	300	16	428	0	0	2	3
PullUpField	DoubleClassChildField	60	1:14	44	0	18	10	6	0	44	1	1
	TripleClassChildField	144	3:06	108	0	42	36	20	0	46	2	2
PullUpMethod	DoubleClassChildMethod	576	14:38	448	0	176	0	48	0	224	0	1
	TripleClassChildMethod	1152	29:08	864	0	336	160	160	0	336	2	2
CS(ChangeReturnType)	MethodReference	3816	37:36	3816	1992	n/a	0	n/a	0	n/a	0	n/a
CS(RemoveParameter)	MethodReference	5724	54:29	5724	1908	0	0	0	0	0	0	0
CS(RemoveParameter)	MethodParamRef.	1680	7:11	772	772	772	0	0	0	0	0	0
MemberToTop	ClassRelationships	70	0:36	51	0	0	0	2	0	2	0	1
	DoubleClassFieldRef.	6600	29:04	2824	0	0	353	507	0	2824	1	1
										Total Bugs:	**21**	**26**

Table 4.3: Summary for ASTGen tests in NetBeans and Eclipse.

4.4.2 Detailed results for relevant ASTGen tests

We provide explanation of our results for the following ASTGen test generators: TripleClassChildField, DoubleClassParentMethod, and DoubleClassGetterSetter. These generators found several bugs in three refactorings. Although we found bugs in NetBeans and Eclipse, we only provide an explanation for bugs related to NetBeans since finding bugs in this IDE is the purpose of this thesis.

TripleClassChildField

This test generator creates tests for the PullUpField refactoring which have the following characteristics:

- Refactoring is: PullUpField B.g to the immediate superclass.
- The tests input programs have 3 classes: A, B, and C. There is an inheritance relationship between the last two classes. Classes A and C declared a field f, and class B declares a field g which optionally references field A.f or field C.f
- There are several ways to arrange classes A, B, and C. For instance, one option is that B is an internal class from A, and class C is a separate class.

Two examples:

We show two tests of TripleClassChildField. Figure 4.3 shows test14, and Figure 4.4 shows test138.

test14, PullUpField B.g to superclass A.C

```
public class A {
  public static boolean f;

  static class C {
    public static boolean f;
  }

}

class B extends A.C {
  boolean g=super.f;
}
```

Figure 4.3: test14 for PullUpField

test138, PullUpField B.g to superclass C

```
public class A {
  public static boolean f;
  public void dummyMethod_A(){
    class B extends C {
      boolean g=A.this.f;
    }
  }
}

class C {
  public static boolean f;
}
```

Figure 4.4: test138 for PullUpField

Explanation of the results:

This test generator creates 144 tests, of which 108 compile. For 66 tests, NetBeans produced a refactored program, and for 42 tests, NetBeans returned an error message. Table 4.4 shows the summary results for this test generator.

		Oracles		
Group of tests	**# Tests**	**NR**	**DNC**	**Diff**
NONE	0	0	n/a	n/a
EC	42	42	n/a	n/a
NB	0	n/a	0	n/a
BOTH	66	n/a	20	46
Totals	108	42	20	46

Table 4.4: Summary for TripleClassChildField

- **Not refactored:**

  There are 42 tests for which NetBeans returned an error message. They correspond to tests where class B is a local class of A. The following error message was returned by NetBeans:

  `- Class B does not exist`

  We manually replicated these tests in NetBeans version 6.0 M9, and the refactoring was correctly performed. Therefore, we did not report this as a bug in Issuezilla. Although this was a bug in NetBeans version 6.0 M3 that we used, it is not any more a bug in NetBeans version 6.0 M9, the latest version as of this writing.

- **Do not compile:**

  There are 20 tests for which NetBeans produced a refactored program that did not compile. They correspond to the following compilation errors:

  - `cannot find symbol:  variable f`

  - `non-static variable this cannot be referenced from a static context`

  - `f has private access in A`

  - `not an enclosing class:  A`

- **Different outputs:**

  There are 46 tests for which NetBeans and Eclipse produced refactored programs with different output files. The difference for these tests is:

  - `fields boolean g = f; and protected boolean g = f; are not equal`

    This difference was not replicated in NetBeans version 6.0 M9. Therefore, we did not report this as a bug in Issuezilla. Although this was a bug in NetBeans version 6.0 M3 that we used, it is not any more a bug in NetBeans version 6.0 M9, the latest version as of this writing.

Bugs found in NetBeans:

We found 2 bugs for PullUpField in NetBeans using TripleClassChildField. They correspond to bugs 108478 and 108479. Table 4.5 shows bug 108478, and Table 4.6 shows bug 108479.

Bug:	#108478
Refactoring:	PullUpField B.g
Problem:	Compilation error: not an enclosing class: A
Solution:	It should return a message such as: "B.g references a field that is not accessible from class C"

Before:

```
A.java
public class A {
    public boolean f;
    class B extends C {
        boolean g = A.this.f;
    }
}

class C {
    public boolean f;
}
```

After:

```
A.java
public class A {
    public boolean f;
    class B extends C {
    }
}

class C {
    public boolean f;
    boolean g = A.this.f;
}
```

Table 4.5: Bug #108478 for PullUpField

Bug: #108479

Refactoring: PullUpField B.g

Problem: Compilation error: f has private access in A

Solution: It should return a message such as: "The visibility of field A.f will be changed to protected"

Before:

```
A.java
public class A {
    private boolean f;
    class B extends C {
        boolean g=new A().f;
    }
}

class C {
    private boolean f;
}
```

After:

```
A.java
public class A {
    private boolean f;
    class B extends C {
    }
}

class C {
    private boolean f;
    boolean g = new A().f;
}
```

Table 4.6: Bug #108479 for PullUpField

DoubleClassParentMethod

This test generator creates tests for the PushDownMethod refactoring which have the following characteristics:

- Refactoring is: PushDownMethod A.m() to immediate subclass.
- The tests input programs have two classes: A and B. There is an inheritance relationship between classes A and B. The superclass A declares a field 'theField' and a method 'm'. Optionally, a method 'mPrime' that references the field 'theField' and method 'm' is declared in class A or class B.
- There are 3 possible ways to arrange classes A and B. For instance, one option is that A and B are separate classes, and B inherits from class A.

Two examples:

We show two tests of DoubleClassParentMethod. Figure 4.5 shows test74, and Figure 4.6 shows test522.

test74, PushDownMethod A.m() to subclass B

```
public class A {
  private Object theField;
  public void m(){
    new A().theField=null;
  }
  void m(  Object newArg){
    A.this.m();
  }
}

class B extends A {
}
```

Figure 4.5: test74 for PushDownMethod

test522, PushDownMethod A.m() to subclass B

```
public class A {
  private Object theField;
  public void m(){
  }
}

class B extends A {
  void m(  Object newArg){
    m();
  }
}
```

Figure 4.6: test522 for PushDownMethod

Explanation of the results:

This test generator creates 960 tests, of which 820 compile. For 520 tests, NetBeans produced a refactored program, and for 300 tests, NetBeans returned an error message. Table 4.7 shows the summary results for this test generator.

		Oracles		
Group of tests	**# Tests**	**NR**	**DNC**	**Diff**
NONE	300	300	n/a	n/a
EC	0	0	n/a	n/a
NB	484	0	412	n/a
BOTH	36	n/a	16	0
Totals	820	300	428	0

Table 4.7: Summary for DoubleClassParentMethod

- **Not refactored:**

  There are 300 tests for which NetBeans returned an error message. The following error message was returned by NetBeans:

  ```
  - Cannot PushDown any members.  The selected type has no subtypes in the currently opened
    projects.
  ```

- **Do not compile:**

  There are 428 tests for which NetBeans produced a refactored program that did not compile. They correspond to the following compilation errors:

  ```
  - cannot find symbol:  method m()
  - not an enclosing class:  A
  - m(java.lang.Object) in A cannot be applied to m()
  ```

- **Different outputs:**

  There were no tests for which NetBeans and Eclipse produced refactored programs with different output files.

Bugs found in NetBeans:

We found 2 bugs for PushDownMethod in NetBeans using DoubleClassParentMethod. They cor-

respond to bugs 108482 and 108484. Table 4.8 shows bug 108482, and Table 4.9 shows bug 108484.

Bug:	#108482
Refactoring:	PushDownMethod A.m()
Problem:	Compilation error: not an enclosing class: A
Solution:	'A.this.theField' should have been changed to 'B.this.theField'

Before:

```
A.java
public class A {
    public Object theField;
    public void m(){
        A.this.theField=null;
    }
}


B.java
class B extends A {
    void mPrime(){
        m();
    }
}
```

After:

```
A.java
public class A {
    public Object theField;
}


B.java
class B extends A {
    void mPrime(){
        m();
    }
    public void m() {
        A.this.theField = null;
    }
}
```

Table 4.8: Bug #108482 for PushDownMethod

Bug:	#108484
Refactoring:	PushDownMethod A.m()
Problem:	Compilation error: cannot find symbol: method m()
Solution:	'new A().m()' should have been changed to 'new B().m()'

Before:

```
A.java
package p;
public class A {
  public void m(){
  }
}

B.java
package p;
class B extends A {
  void mPrime(){
    new A().m();
  }
}
```

After:

```
A.java
package p;
public class A {
}

B.java
package p;
class B extends A {
  void mPrime(){
    new A().m();
  }
  public void m() {
  }
}
```

Table 4.9: Bug #108484 for PushDownMethod

DoubleClassGetterSetter

This test generator creates tests for the EncapsulateField refactoring which have the following characteristics:

- Refactoring is: EncapsulateField A.theField
- The test input programs have two classes: A and B. There is an inheritance relationship between classes A and B. The superclass A declares a field 'theField', subclass B has either a setter or getter for 'theField', and superclass A has a main method that invokes either the setter or getter for 'theField'.
- There are 3 possible ways to arrange classes A and B. For instance, one way is that B is an internal class of class A.
- The setter method can have different return types.

Two examples:

We show two tests of DoubleClassGetterSetter. Figure 4.7 shows test4, and Figure 4.8 shows test60.

test4, EncapsulateField A.theField

```
public class A {
  public int theField;

  public static void main(String[] args){
    B b=new A().new B();
    b.setTheField(10);
  }

  class B extends A {
    public int setTheField(int theField){
      this.theField=theField;
      return 2;
    }
  }
}
```

Figure 4.7: test4 for EncapsulateField

test60, EncapsulateField A.theField

```
public class A {
  public String theField;

  public static void main(  String[] args){
    B b=new A().new B();
    System.out.println(b.getTheField());
  }

  class B extends A {
    private String getTheField(){
      return theField;
    }
  }
}
```

Figure 4.8: test60 for EncapsulateField

Explanation of the results:

This test generator creates 576 tests, of which 417 compile. For all tests, NetBeans produced a refactored program. Table 4.10 shows the summary results for this test generator.

		Oracles			
Group of tests	**# Tests**	**NR**	**DNC**	**Diff**	**COB**
NONE	0	0	n/a	n/a	0
EC	0	n/a	n/a	n/a	n/a
NB	216	0	n/a	n/a	0
BOTH	201	0	162	0	39
Totals	417	0	162	0	39

Table 4.10: Summary for DoubleClassGetterSetter

- **Not refactored:**

  NetBeans generated a refactored program for each test.

- **Do not compile:**

  There are 162 tests for which NetBeans produced a refactored program that did not compile. They correspond to the following compilation errors:

  - `setTheField(int) in A.B cannot override setTheField(int) in A; attempting to assign weaker access privileges; was public`
  - `getTheField() in A.B cannot override getTheField() in A; attempting to assign weaker access privileges; was public`
  - `setTheField(int) in A.B cannot override setTheField(int) in A; attempting to use in compatible return type`

- **Different outputs:**

  There were no tests for which NetBeans and Eclipse produced refactored programs with different output files.

- **Different behavior:**

  There are 39 tests for which NetBeans produced a refactored program with changed behavior. They correspond to the following changes of behavior:

  - `Exception in thread 'main' java.lang.StackOverflowError at A\B.setTheField(A.java:22)`
  - `Exception in thread 'main' java.lang.StackOverflowError at A\B.getTheField(A.java:1)`

Bugs found in NetBeans:

We found 3 bugs for EncapsulateField in NetBeans using DoubleClassGetterSetter. They correspond to bugs 108473, 108474, and 108489. Table 4.11 shows bug 108474, and Table 4.12 shows bug 108489.

Bug:	#108474
Refactoring:	EncapsulateField A.theField
Problem:	Change of behavior. Running class A before the refactoring terminates normally and returns some value, where as the other run results in this exception: 'main' java.lang.StackOverflowError at B.setTheField(A.java:22)
Solution:	It should return an error message such as: "This refactoring will produce a change of behavior"

Before:

```
A.java
public class A {
  public String theField;

  public static void main(String[] args){
    B b=new B();
    b.setTheField("abcd");
  }
}

class B extends A {
  public void setTheField(String theField){
    this.theField=theField;
  }
}
```

After:

```
A.java
public class A {
  private String theField;

  public static void main(String[] args){
     B b=new B();
     b.setTheField("abcd");
  }
  public String getTheField() {
     return theField;
  }
  public void setTheField(String theField) {
     this.theField = theField;
  }
}
class B extends A {
    public void setTheField(String theField){
        this.setTheField(theField);
    }
}
```

Table 4.11: Bug #108474 for EncapsulateField

Bug: #108489
Refactoring: EncapsulateField A.f
Problem: Compilation error: setF(String) in B cannot override setF(String) in A; attempting to use incompatible return type
Solution: It should return an error message: "Overridden methods in subclasses must have the same return type"

Before:

```
A.java
public class A {
    String f;
    public static void main(String[] args){
        B b=new B();
        b.setF("abcd");
    }
}
class B extends A {
    public int setF(String theField){
        this.f=theField;
        return 2;
    }
}
```

After:

```
A.java
public class A {
    private String f;
    public static void main(String[] args){
        B b=new B();
        b.setF("abcd");
    }
    public String getF() {
        return f;
    }
    public void setF(String f) {
        this.f = f;
    }
}
class B extends A {
    public int setF(String theField){
        this.setF(theField);
        return 2;
    }
}
```

Table 4.12: Bug #108489 for EncapsulateField

4.5 Discussion

NetBeans and Eclipse refactoring engines perform a refactoring even when the input program does not compile. We consider that refactorings should have as a general precondition that the input program compiles to avoid applying program transformations over incorrect input programs.

Writing the test generators in ASTGen was relatively easy because we reuse some basic generators included in its library. While we can in theory write only one, very generic test generator for each refactoring, it is convenient in practice to have a list of different generators that focus on different scenarios for each refactoring taking into account its preconditions and the Java syntax. Otherwise, we could generate similar tests with various test generators.

In order to find some bugs related to change of behavior, it is necessary to exercise different paths for an input program that includes a main method. It would be interesting to generate tests for the program being refactored such that we can check whether the behavior remains the same.

Chapter 5

Related work

This chapter describes some work related to this thesis. It first discusses differential testing and then test generation.

Differential testing has successfully been applied before to test software such as programming language compilers [7]. It requires two or more comparable programs and some test inputs. These test inputs are given to each program and their results are compared to check for differences and thus potential bugs.

Differential testing can be used to measure the quality of a software [7]. It is recommended to use differential testing when the objective is to increase the quality of a stable piece of software, to replace an old software, or when new software is challenging a competitor [7]. Therefore, differential testing is applicable to NetBeans and Eclipse as they are stable pieces of software. There are three general issues that have to be addressed to use differential testing effectively: (1) using quality tests that exercise different paths, (2) identifying false positives which are differences due to different implementations, and (3) handling the amount of redundant tests among generated tests. One advantage of differential testing is that it provides a solution for the 'oracle problem', the problem of evaluating test results. This is particularly useful when we use automatically generated tests.

There is a large body of work in the area of test-input generation. The most closely related to ASTGen are grammar-based and bounded-exhaustive testing approaches.

Grammar-based testing [15, 16, 17, 18, 19] requires the user to describe test inputs with a grammar, and the tools then generate a set of strings that belong to the grammar (or sometimes a set

of strings that intentionally do not belong to the grammar). In 1972, Purdom [15] pioneered the algorithms for selecting a minimal set of strings that achieve certain coverage criteria for grammars, e.g., strings that cover all terminals, all non-terminals, or all productions. More recently, Maurer [18], Sirer and Bershad [17], and Malloy and Power [19] developed tools for grammar-based generation that were used to find bugs in several applications. We can view grammar-based approaches as effectively using first-order functional programs to specify the generation. The tools interpret these programs typically to generate *random* strings that belong to the grammar.

In contrast to random generation, the approach of Lämmel and Schulte [16] and ASTGen *systematically* generate input data, which can often catch "corner cases" that random testing misses. Lämmel and Schulte base their approach on grammars and provide several parameters with which testers can control the "exhaustiveness" of generation of strings from the grammar. ASTGen allows testers to use the full expressive power of a familiar programming language such as Java to write *imperative generators* that produce test inputs.

With ASTGen approach, testers can freely compose more basic generators into more advanced generators. Achieving such reusability with grammars is fairly hard. For example, it is not obvious how, in a grammar-based approach, one could combine the following two generators to obtain a new generator: (1) a generator that produces two classes that exhibit all possible relationships such as class inheritance, containment, or class name reference and (2) a generator that produces classes with field references. In addition, dependent composition becomes difficult or impossible with grammars. It has been long realized that even the simplest cases require extensions to the grammar, e.g., the use of attributed grammars [20], and to generate valid programs as inputs (e.g., to test compilers) requires even further extensions to context-free grammars [21, 22].

ASTGen framework for imperative generation was inspired by QuickCheck [23], a Haskell library for random generation of test data. QuickCheck provides a set of basic generators (each of which is a Haskell monad) and combinators for building complex generators from simpler ones. ASTGen framework uses Java classes instead of Haskell monads and provides bounded-exhaustive genera-

tion. In ASTGen, both generators and their composition are expressed in Java, but the ASTGen developers plan to consider other approaches for composition such as GenVoca [24].

Bounded-exhaustive testing [10, 11, 12, 13, 14] is an approach for testing code exhaustively on all inputs within the given bound. Two previously developed approaches, TestEra [11] and Korat [13], can in principle be used for bounded-exhaustive generation of complex test inputs such as Java programs. These two approaches are *declarative*: they require the user to specify the constraints that describe *what* the test inputs look like (as well as the bound on the size of test inputs), and the tools then automatically search the (bounded) input space to generate all inputs that satisfy the constraints. TestEra requires the user to specify the constraints in a declarative language, while Korat requires the users to specify the constraints in an imperative language. In both previous approaches, the user just specifies the constraints.

The ASTGen approach differs in that it is *imperative*: the programmer specifies *how* the test generation should proceed. The imperative approach makes the generation faster since no search is necessary. Also, the imperative approach gives the programmer more control over the generation, for example over the order of generation. Finally, the two previous declarative approaches have not been applied to generate inputs as complex as Java programs, whereas ASTGen have been applied to generate Java programs to test refactoring engines in NetBeans and Eclipse.

Nevertheless, the imperative approach has some disadvantages compared to the declarative approach. The declarative approach may be more appealing when the testers are not willing to write generators or the constraints are fairly simple. Also, the declarative approach always generates valid inputs, whereas most of our imperative generators rely on the compiler to filter valid inputs, which can reduce performance of generation for larger ASTs. It remains as future work to empirically compare these two approaches.

Chapter 6

Conclusions

Refactoring engines have become popular because they allow programmers to quickly and (for the most part) safely change large programs. These tools also influence the culture of software development: programmers who use refactoring engines are more inclined to change large programs. Despite the high quality and widespread use of existing refactoring engines, they still contain bugs. Our goal is to help developers of NetBeans refactoring engine to find bugs.

This thesis presented our work on testing the refactoring engine in the NetBeans development environment. We tested the following six refactorings in NetBeans: Rename, ChangeMethodParameters, EncapsulateFields, PullUp, PushDown, and MoveClassInnerToOuterLevel. We found 53 bugs: 27 bugs were found using Eclipse tests, and 26 bugs were found using ASTGen tests. We reported these bugs in Issuezilla, and the developers have already fixed 5 of these bugs, declared 7 bugs as duplicates, declared 1 bug as "won't be fixed", declared 2 bugs as "work fine in latest version", and confirmed 43 as real bugs.

As future work, we list the following topics:

- Reducing the number of redundant tests

  The ASTComparator compares NetBeans and Eclipse output files and generates a `txt` file that lists the differences. We manually had to inspect the differences to find bugs. This part was time consuming because there were more than 400 differences for some ASTGen test generators. In most cases, there were several tests related to the same bug; this was particularly true for generated tests. In that sense, reducing the number of redundant tests contributes to the scalability of our solution. We can group tests based on their error messages, compilation errors,

and differences between output files. For instance, two pairs of output files where each pair differs only in a field visibility modifier (e.g. "boolean g=true" and "protected boolean g=true" vs. "boolean g" and "protected boolean g") be interpreted as the same. We can also use test reduction techniques for each group of tests to find a smaller test that exposes the same bug since developers often prefer a smaller test.

- Reducing the number of false alarms

  The Not refactored Oracle and Different outputs Oracle produce some false positives. For Not refactored, the reason is that some generators produce programs (1) for which the refactoring is expected to find preconditions violations and (2) for which the refactoring is expected to proceed and refactor the input program. One could reduce this problem by writing generators that produce only one or the other kind of program. However, we found that refactoring engines can have different preconditions for some refactorings. For Different outputs, the reason is that the ASTComparator sometimes finds two programs different even though they are semantically equivalent at the sub-method level. While the ASTComparator does compare the programs purely syntactically, it does not (and cannot) compare their full semantics. Additionally, Different outputs Oracle can be triggered by the different ways in which the two engines perform refactorings. For example, when renaming a method m, one engine may also rename all overloaded methods, while the other may not. Neither approach is semantically incorrect, so there is no bug. It is required to better account for semantics and engine differences to reduce or eliminate the number of false positives.

- Coverage for refactoring engines

  Using coverage tools, we can get coverage information for NetBeans refactoring engine using Eclipse tests and automatically generated tests. This coverage can be used to guide generation of new tests.

- Make refactoring runners easier

  We used NetBeans version 6.0 M3 to run the refactorings, and we had to make a number of changes to run Eclipse and ASTGen tests in NetBeans. We are in contact with the NetBeans developers, and their latest versions will probably address these changes.

- Add more ASTGen tests

  More ASTGen generators can be written to test refactorings. As was mentioned before, a list of different scenarios for each refactoring can be very useful to avoid that several ASTGen generators produce similar tests.

- Test more refactorings

  It is desirable to test more refactorings. They can be prioritized using the NetBeans and Eclipse version that first provided them, the number of bugs reported in Issuezilla and Bugzilla, or any other criteria.

References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison Wesley Professional, 1999.

[2] D. B. Roberts, "Practical analysis for refactoring," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1999, http://st-www.cs.uiuc.edu/~droberts/thesis.pdf.

[3] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[4] D. Gallardo, "Refactoring for everyone," September 2003, http://www-128.ibm.com/developerworks/library/os-ecref/.

[5] W. F. Opdyke and R. E. Johnson, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems," in *Proc. Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, September 1990.

[6] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Symposium on the Foundations of Software Engineering (ESEC/FSE)*, September 2007.

[7] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, 1998.

[8] R. Enns, "Refactoring in Eclipse," University of Manitoba, Canada, 2004, http://www.cs.umanitoba.ca/~eclipse/13-Refactoring.pdf.

[9] J. Prox, 2007, personal email communication.

[10] S. Khurshid, "Generating structurally complex tests from declarative constraints," Ph.D. dissertation, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2003.

[11] S. Khurshid and D. Marinov, "TestEra: Specification-based testing of Java programs using SAT," *Automated Software Engineering Journal*, vol. 11, no. 4, pp. 403–434, October 2004.

[12] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson, "Software assurance by bounded exhaustive testing," in *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, Boston, MA, July 2004.

[13] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on Java predicates," in *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, Rome, Italy, July 2002.

[14] D. Marinov, "Automatic testing of software with structurally complex inputs," Ph.D. dissertation, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2004.

[15] P. Purdom, "A sentence generator for testing parsers," *Behavior and Information Technology*, vol. 12, no. 3, pp. 366–375, 1972.

[16] R. Lämmel and W. Schulte, "Controllable combinatorial coverage in grammar-based testing," in *The 18th IFIP International Conference on Testing Communicating Systems (TestCom)*, May 2006, pp. 19–38.

[17] E. G. Sirer and B. N. Bershad, "Using production grammars in software testing," in *2nd Conference on Domain-specific Languages*, Austin, TX, October 1999.

[18] P. M. Maurer, "Generating test data with enhanced context-free grammars," *IEEE Software*, vol. 7, no. 4, 1990.

[19] B. A. Malloy and J. F. Power, "An interpretation of Purdom's algorithm for automatic generation of test cases," *1st Annual International Conference on Computer and Information Science*, October 2001.

[20] A. G. Duncan and J. S. Hutchison, "Using attributed grammars to test designs and implementations," in *Proc. of the 5th International Conference on Software Engineering (ICSE)*, San Diego, CA, March 1981, pp. 170–178.

[21] F. Bazzichi and I. Spadafora, "An automatic generator for compiler testing." *IEEE Trans. Software Eng.*, vol. 8, no. 4, pp. 343–353, July 1982.

[22] A. Celentano, S. C. Reghizzi, P. D. Vigna, C. Ghezzi, G. Granata, and F. Savoretti, "Compiler testing using a sentence generator," *Software - Practice and Experience*, vol. 10, no. 11, pp. 897–918, 1980.

[23] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of Haskell programs," in *Fifth ACM SIGPLAN International Conference on Functional Programming*, Montreal, CA, September 2000, pp. 268–279.

[24] D. S. Batory and S. W. O'Malley, "The design and implementation of hierarchical software systems with reusable components," *ACM Transactions on Software Engineering Methodology*, vol. 1, no. 4, pp. 355–398, October 1992.